Kaappi

A Scheme Programming Language

Baiju Muthukadan

R7RS-small • Zig • Batteries Included

Kaappi: A Scheme Programming Language

Baiju Muthukadan

First Edition, 2026

Feedback and errata reports are welcome at baiju.m.mail@gmail.com.

Contents

Preface

Why Scheme?

If you already know Python, JavaScript, or Ruby, you can write programs that work. So why learn another language—especially one with all those parentheses?

Scheme will change the way you think about programming. Its syntax is so minimal that you can hold the entire language in your head. There are no special forms for loops, no operator precedence rules, no class hierarchies. Instead, you get a small set of powerful ideas—first-class functions, lexical closures, recursion, and macros—that combine to express anything you need.

This matters because Scheme's ideas are everywhere. Python borrowed `lambda`, list comprehensions, and garbage collection from Lisp. JavaScript's closures and first-class functions come directly from Scheme. Ruby's blocks are a limited form of Scheme's closures. Learning Scheme means understanding the foundations that these languages are built on.

Why Kaappi?

Kaappi is a modern, complete implementation of the R7RS-small standard. Unlike many Scheme implementations that focus on either academic purity or raw performance, Kaappi aims to be a practical, batteries-included language for building real programs.

It ships as a statically linked binary with no shared library dependencies—just the binary and a library archive of portable Scheme files. The interactive REPL provides syntax highlighting, tab completion, and history—a comfortable environment for exploration. Beyond the standard, Kaappi includes 72 SRFIs (8 compiled into the binary, 64 as portable Scheme libraries), a C foreign function interface, green threads with channels, a package manager, and an ecosystem of libraries for web servers, databases, JSON, and more.

If you have ever tried a Scheme and felt it was "just a toy," Kaappi is different. You can build real applications with it.

Who This Book Is For

This book is for programmers who know at least one mainstream language—Python, JavaScript, Ruby, Java, Go, or similar—and want to learn Scheme. No prior experience with Lisp or functional programming is required.

This is not an academic textbook. You will not find formal semantics or denotational models here. Instead, you will find clear explanations, running code, and practical exercises. Theory is introduced only when it helps you write better programs.

What You Will Need

- **Kaappi installed on your system.** Chapter 2 covers installation in detail. The short version: `curl -fsSL https://kaappi-lang.org/install.sh | bash`
- **A terminal.** You will spend most of your time in the Kaappi REPL, typing expressions and seeing results immediately.
- **A text editor.** Any editor works. If you use VS Code, Neovim, or Emacs, Kaappi provides an LSP server for go-to-definition, find-references, and document symbols.

How to Use This Book

The book is organized into four parts:

Part I: Getting Started covers installation, the REPL, and basic expressions. Start here.

Part II: The Language teaches the core of Scheme: data types, lists, control flow, functions, bindings, higher-order functions, and I/O. This is the heart of the book.

Part III: Going Further introduces macros, the library system, records, error handling, and continuations. Read this once you are comfortable writing functions and working with lists.

Part IV: Beyond R7RS explores Kaappi-specific features: the SRFI ecosystem, the C foreign function interface, and concurrency with fibers and threads.

Each chapter builds on the previous ones. If you are new to Scheme, work through them in order. If you already know some Scheme and want to learn what Kaappi offers beyond the standard, skip to Part IV.

The appendices are reference material to dip into as needed: an R7RS quick reference, a command-line reference, the full SRFI catalog, common error messages and their fixes, the ecosystem package list, a glossary, and pointers for further reading. If you hit an unfamiliar term while reading, the Glossary (Appendix F) is the fastest place to look it up.

The most effective way to use this book is with the Kaappi REPL open beside it. Type every example. Modify them. Break them. The REPL gives instant feedback, and that tight loop between reading and trying is how Scheme clicks.

Exercises appear at the end of most chapters. They are not optional filler—they are where you solidify your understanding. Attempt them before moving on.

This book does not print exercise solutions, and that is deliberate. Scheme makes it easy to check your own work: type your answer into the REPL, compare its output against the expected result shown in the exercise, and refine until they match. Many exercises are open-ended—there is no single correct answer, and the point is to build fluency rather than match a key. If you get truly stuck, reread the relevant section with the REPL open and experiment until the behavior makes sense.

Conventions

Throughout this book:

Code listings show Scheme source code with syntax highlighting. Keywords are bold.

REPL sessions show interactive input (lines starting with >) and output. Type the input lines into your Kaappi REPL to follow along.

Notes provide supplementary information, tips, or comparisons to Python.

Warnings highlight common mistakes and pitfalls.

Summaries recap a chapter's key points in a short bulleted list at the end, just before the exercises.

Exercises are numbered per chapter and provide practice problems of increasing difficulty.

Part I

Getting Started

This part takes you from nothing to a working setup. You will see what Scheme and Kaappi are, install the interpreter, get comfortable in the REPL, and write your first expressions and functions. By the end you will be ready to explore the language in depth.

Chapter 1
Introduction

1.1 What Is Scheme?

Scheme is a programming language in the Lisp family, created at MIT in 1975 by Gerald Jay Sussman and Guy L. Steele Jr. They wanted a language small enough to understand completely, yet powerful enough to express any computation elegantly.

They succeeded. Scheme's core fits on a single page: variables, functions, conditionals, and a handful of special forms. From these primitives, everything else is built—loops, objects, modules, pattern matching, even new syntax. The language has no reserved words that you cannot redefine, no special cases for "built-in" constructs.

Scheme's key ideas:

- **Minimalism.** The language has very few rules. Once you learn them, there are no surprises.
- **First-class procedures.** Functions are values, just like numbers and strings. You can pass them to other functions, return them, and store them in data structures.
- **Lexical scope.** A function remembers the environment where it was defined. This creates closures—one of the most powerful ideas in programming.
- **Proper tail recursion.** Recursive functions that call themselves in tail position use constant stack space. Loops are just a pattern of recursion.
- **Continuations.** You can capture "the rest of the computation" as a value and invoke it later. This enables non-local exits, coroutines, and backtracking without special language support.

- **Hygienic macros.** You can define new syntactic forms that look and behave like built-in ones, without variable-capture bugs.

The language has been standardized several times over its five-decade history:

Year	Standard	Significance
1975	Original paper	Sussman & Steele at MIT
1991	R4RS	First widely adopted standard
1998	R5RS	The "classic" standard; still popular
2007	R6RS	Larger standard; controversial in the community
2013	R7RS-small	Current standard; returns to minimalism

R7RS-small is what this book teaches. It defines the syntax, semantics, and a set of standard libraries that any conforming implementation must provide.

1.2 Scheme vs. Python: A First Glimpse

If you know Python, Scheme will look alien at first. But the differences are more superficial than they appear. Let's compare a few simple tasks.

This book uses Python as its comparison language throughout. If you come from JavaScript or Ruby instead, the parallels carry over almost unchanged—JavaScript's arrow functions and Ruby's blocks play the same role that Python's functions and `lambda` play in these comparisons.

Hello, World

In Python:

```
print("Hello, world!")
```

In Scheme:

```
(display "Hello, world!")
(newline)
```

The parentheses mean "apply this procedure to these arguments." `display` is Scheme's equivalent of Python's `print`. The procedure name comes first—always.

A Recursive Function

Factorial in Python:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Factorial in Scheme:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

The structure is the same—a base case and a recursive case. Scheme just uses prefix notation and parentheses instead of indentation and keywords.

Filtering a List

In Python:

```
evens = [x for x in range(10) if x % 2 == 0]
```

In Scheme:

```
(define evens
  (filter even? (iota 10)))
```

`filter` takes a predicate and a list, returning elements that satisfy the predicate. `iota` generates a list of numbers (like Python's `range`). `even?` is a built-in predicate—note the question mark, a Scheme convention for predicates. Both `filter` and `iota` come from SRFI-1, a widely used list library that Kaappi builds in, so they work out of the box. Portable R7RS code declares the dependency with `(import (srfi 1))`—a convention Chapter 12 explains. Do not worry about mastering these now—Chapter 9 covers `filter` and its relatives in depth.

Closures

A function that returns a function—a *closure*—works the same way in both languages.

In Python:

```
def make_adder(n):
```

```
    return lambda x: x + n

add5 = make_adder(5)
add5(3)  # => 8
```

In Scheme:

```
(define (make-adder n)
  (lambda (x) (+ x n)))

(define add5 (make-adder 5))
(add5 3)  ; => 8
```

The inner function remembers `n` from its enclosing scope. This pattern is fundamental in Scheme—it replaces classes, objects, and much of what other languages need special syntax for.

Error Handling

In Python:

```
try:
    result = risky_operation()
except ValueError as e:
    result = default_value
```

In Scheme:

```
(guard (e ((error-object? e) default-value))
  (risky-operation))
```

Scheme's `guard` is structurally similar to Python's `try/except`, but it is an expression that produces a value.

> **Coming from Python.** The biggest adjustment is prefix notation: the operator always comes first. Instead of `2 + 3`, you write `(+ 2 3)`. Instead of `f(x, y)`, you write `(f x y)`. There is no operator precedence to memorize—parentheses make the grouping explicit.

1.3 What Is Kaappi?

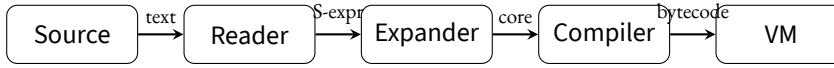
Kaappi is a complete implementation of the R7RS-small Scheme standard, written from scratch in Zig. The name comes from South Indian languages such as Malayalam and Tamil, where *kaappi* means "coffee"—a nod to the warm, everyday ritual of brewing something energizing from simple ingredients.

Standards Conformance

Kaappi implements every identifier in the R7RS-small specification: all 14 standard libraries, all 32 syntax forms, and 613 built-in procedures. It passes 1,391 tests from the R7RS test suite with zero failures. When the standard says how something should work, that is exactly how Kaappi behaves.

How It Works

When you type a Scheme expression, Kaappi processes it through a pipeline:



1. **Reader** — parses source text into a data structure (an S-expression).
2. **Expander** — processes macros, transforming syntactic sugar into core forms.
3. **Compiler** — translates the expanded code into register-based bytecode through an intermediate representation with optimization passes.
4. **Virtual machine** — executes the bytecode on a register-based VM with a generational mark-and-sweep garbage collector.

This architecture means Kaappi is fast enough for real work while remaining simple enough that the entire implementation is readable (about 48,000 lines of Zig).

Batteries Included

Beyond the R7RS standard, Kaappi provides:

- **72 SRFIs** — community-standard libraries for lists, strings, hash tables, sorting, regular expressions, random numbers, and much more.

- **C FFI** — call functions in C shared libraries directly from Scheme, including passing Scheme callbacks into C code.
- **Green threads** — lightweight fibers with channels for concurrent programming without the complexity of OS threads.
- **Package manager** — `thottam` installs and manages third-party libraries with a single command.
- **Ecosystem libraries** — HTTP clients and servers, PostgreSQL and SQLite drivers, JSON and TOML parsers, HTML templates, email, cryptography, logging, and CLI frameworks.
- **Developer tools** — an LSP server for editor integration, a stepping debugger, a profiler, and a disassembler.

Single Binary, Multiple Platforms

Kaappi installs as a statically linked binary plus a small archive of portable Scheme libraries—no shared library dependencies, no runtime to configure. It runs on macOS (ARM64), Linux (x86_64, ARM64, RISC-V), and in the browser via WebAssembly.

1.4 Why Learn Scheme?

There are practical reasons to learn Scheme, even if you do not plan to use it as your primary language.

It teaches you to think differently. Scheme forces you to solve problems with recursion and composition rather than mutation and iteration. Once this way of thinking clicks, you will write better code in any language—your Python will use more generators and less manual state management, your JavaScript will leverage closures instead of class hierarchies.

Macros give you superpowers. In most languages, you are limited to the abstractions the language designers provided. In Scheme, you can extend the language itself. If you find yourself writing the same boilerplate pattern over and over, you can define a macro that eliminates it—and the macro looks just like a built-in form.

It makes you a better programmer. Understanding lexical scope, closures, continuations, and tail recursion at a deep level gives you insight into how all languages work under the hood. Scheme strips away the incidental complexity (classes, decorators, type annotations, build systems) and lets you focus on the essential complexity of the problem you are solving.

Kaappi makes it practical. With a package manager, database drivers, web frameworks, and a C FFI, you are not limited to academic exercises. You can build CLI tools, REST APIs, data pipelines, and automation scripts in Scheme.

1.5 How This Book Differs

Most Scheme books fall into one of two camps: academic textbooks that use Scheme to teach computer science concepts (like SICP), or reference manuals that document every procedure without showing how to build real programs.

This book is neither. It is a *practical programming guide* aimed at working developers. Every concept is motivated by a concrete problem, demonstrated with running code, and followed by exercises you can type into the REPL. Theory appears only when it helps you write better programs.

The book is also specific to Kaappi. While the core language is standard R7RS, the later chapters cover Kaappi-specific features—the FFI, fibers, the package manager, and the ecosystem libraries—that let you build real applications.

1.6 What You Will Build

By the end of this book, you will be able to write recursive algorithms that operate in constant stack space, define hygienic macros that extend the language, organize code into reusable libraries, handle errors gracefully, call C functions from Scheme, and write concurrent programs using fibers and channels. More importantly, you will understand *why* these features exist and when to reach for each one.

Let's get started.

Summary

- Scheme is a minimalist Lisp dialect built from a few powerful ideas: first-class functions, lexical closures, recursion, and macros.
- Many features of Python, JavaScript, and Ruby—closures, `lambda`, garbage collection—trace back to Lisp and Scheme.
- Kaappi is a practical, batteries-included implementation of R7RS-small, written in Zig and shipped as a single statically linked binary.
- Beyond the standard, Kaappi adds 72 SRFIs, a C foreign function interface, green-threaded concurrency, and a package manager.

Exercises

Exercise 1.1: Try the Playground

Visit `https://kaappi-lang.org/playground/` in your browser. Try evaluating a few expressions from this chapter—the factorial function, the list filter example, or anything else that interests you. The playground runs the same Kaappi interpreter via WebAssembly, so everything works identically to a local install.

Exercise 1.2: Install Kaappi

Install Kaappi on your system using the install script or a manual download (see Chapter 2 for details). Run `kaappi --version` in your terminal and confirm you see a version string like `Kaappi Scheme v0.15.0`. Then evaluate a simple expression:

```
> (+ 1 2 3)
6
```

If you see `6`, your installation is working.

Exercise 1.3: Scheme's Influence on Python

Many features that Python programmers take for granted were pioneered by Lisp and Scheme. Research and list at least three Python features that trace their origins to Lisp or Scheme. Hints: think about garbage collection, first-class functions (lambda), list comprehensions (derived from `map` and `filter`), decorators (higher-order functions), and generators (related to continuations). For each one, write a sentence explaining the connection.

The next chapter covers installing Kaappi and getting comfortable in the REPL.

Chapter 2
Installation and Setup

This chapter gets Kaappi running on your system, introduces the interactive REPL, and shows you how to run script files. By the end, you will have a working environment for the rest of the book.

2.1 Installing Kaappi

Install Script (Recommended)

The fastest way to install Kaappi is the one-line install script. It detects your platform, downloads the correct binary, verifies SHA256 checksums, and places everything in the right location:

```
curl -fsSL https://kaappi-lang.org/install.sh | bash
```

This installs two binaries to `~/.local/bin/`:

- `kaappi` — the Scheme interpreter, compiler, and REPL.
- `thottam` — the package manager for installing third-party libraries.

Standard libraries (SRFIs and Scheme base libraries) are extracted to `~/.kaappi/lib/`.

To install to a different location, set the `INSTALL_DIR` environment variable:

```
curl -fsSL https://kaappi-lang.org/install.sh \
  | INSTALL_DIR=/usr/local/bin bash
```

PATH configuration. Make sure `~/.local/bin` is in your shell's `PATH`. If running `kaappi` after installation gives "command not found," add this line to your `~/.bashrc`, `~/.zshrc`, or equivalent:

```
export PATH="$HOME/.local/bin:$PATH"
```

Then restart your terminal or run `source ~/.zshrc`.

Manual Download

If you prefer not to pipe a script to your shell, download the binary directly from the GitHub releases page. Kaappi provides pre-built binaries for these platforms:

OS	Architecture	Binary name
macOS	ARM64 (Apple Silicon)	`kaappi-aarch64-macos`
Linux	x86_64	`kaappi-x86_64-linux`
Linux	ARM64	`kaappi-aarch64-linux`
Linux	RISC-V 64	`kaappi-riscv64-linux`

Windows. There are no native Windows binaries—Kaappi depends on POSIX APIs. On Windows, install the Linux x86_64 build inside WSL2 (Windows Subsystem for Linux); everything in this book works there.

Each release also ships a matching `thottam` package-manager binary per platform (e.g., `thottam-aarch64-macos`)—download it alongside `kaappi`. After downloading, make the binaries executable and move them to a directory in your PATH:

```
chmod +x kaappi-aarch64-macos thottam-aarch64-macos
mv kaappi-aarch64-macos ~/.local/bin/kaappi
mv thottam-aarch64-macos ~/.local/bin/thottam
```

You also need the standard library archive. Download `kaappi-lib.tar.gz` and extract it:

```
mkdir -p ~/.kaappi/lib
tar xzf kaappi-lib.tar.gz -C ~/.kaappi/lib
```

Building from Source

Building from source requires Zig 0.16 or later. No other build tools (make, cmake, cargo) are needed.

On macOS:

```
brew install zig
git clone https://github.com/kaappi/kaappi
cd kaappi
zig build
```

On Linux, download Zig from `ziglang.org/download`, add it to your PATH, then:

```
git clone https://github.com/kaappi/kaappi
cd kaappi
zig build
```

The binary appears at `zig-out/bin/kaappi`. Copy it to a location in your PATH:

```
cp zig-out/bin/kaappi zig-out/bin/thottam ~/.local/bin/
```

> **Build modes.** The default build uses ReleaseSafe (fast execution with bounds checking). For maximum performance, use `zig build -Doptimize=ReleaseFast`. Avoid Debug mode for normal use—it is roughly 500x slower on allocation-heavy workloads.

2.2 Verifying Your Installation

Check that Kaappi is installed correctly:

```
kaappi --version
```

You should see output like `Kaappi Scheme v0.15.0`. Next, test that evaluation works:

```
echo '(+ 1 2)' | kaappi
```

This should print `3`. If both commands succeed, your installation is ready.

When something is not working—or you simply want a thorough once-over—run the built-in self-check:

```
kaappi doctor
```

`doctor` inspects six areas (the binary, the library search path, the package manager, the native-compiler toolchain, the REPL, and the FFI) and prints `PASS`, `WARN`, or `FAIL` for each check, with a suggested fix under every failure:

```
binary
  PASS  version: v0.15.0
  PASS  target: aarch64-macos-none
  PASS  on-PATH: /Users/ada/.local/bin/kaappi

library
  PASS  ~/.kaappi/lib: /Users/ada/.kaappi/lib
  ...
```

Keep `kaappi doctor` in mind for later chapters too: it is the fastest way to diagnose a missing library path or an FFI toolchain problem.

> **No install yet?.** If you want to follow along before installing, Kaappi provides a browser-based playground at `https://kaappi-lang.org/playground/` that runs the same interpreter via WebAssembly. It is perfect for trying examples, though you will eventually want a local install for file I/O and the full feature set.

2.3 The REPL

The REPL (Read-Eval-Print Loop) is where you will spend most of your time while learning. Launch it by running `kaappi` with no arguments:

```
kaappi
```

You will see a welcome banner and a prompt:

```
Kaappi Scheme v0.15.0
Type ,help for commands, ,quit to exit.

kaappi> (+ 2 3)
5
kaappi> (* 6 7)
42
```

Prompt convention. The actual prompt is `kaappi>`. To keep examples compact, the rest of this book typesets it as a plain `>`.

Multi-Line Input

The REPL tracks open parentheses. If your expression is incomplete, it shows a `...` continuation prompt and waits for you to finish:

```
> (define (square x)
...   (* x x))
> (square 7)
49
```

Line Editing

The REPL supports standard editing keys:

Key	Action
Left / Right	Move cursor
Up / Down	Navigate history
Ctrl-A	Move to start of line
Ctrl-E	Move to end of line
Ctrl-K	Delete to end of line
Ctrl-C	Cancel current line
Tab	Auto-complete symbol names
Ctrl-D	Exit (on empty line)

Tab Completion

Press Tab to complete symbol names. If you type `string-` and press Tab, the REPL shows all matching procedures:

```
> string-
string-append  string-copy  string-length  string-ref  ...
```

Completion works for built-in procedures, your own definitions, and imported library exports.

Syntax Highlighting

As you type, the REPL colors different elements—keywords, strings, numbers, comments—and highlights the parenthesis that matches the one under your cursor. This makes it easier to spot typos and unbalanced parentheses.

The colors are configurable through `~/.kaappi/config`, a plain `key: value` file. `repl.theme` selects the `dark` (default) or `light` preset, `repl.color.token` overrides individual token colors, and `repl.prompt` replaces the prompt text:

```
# ~/.kaappi/config
repl.theme: light
repl.color.string: cyan
repl.prompt: scheme>
```

Setting the `NO_COLOR` environment variable disables all coloring, and `repl.highlight: false` turns highlighting off from the config file.

The _ Variable

The special variable _ always holds the result of the last evaluated expression:

```
> (+ 10 20)
30
> (* _ 2)
60
```

This is useful for building on previous results without re-typing them.

History

Command history is saved to `~/.kaappi/history` and persists across sessions. Use the Up and Down arrow keys to navigate previous inputs.

Exiting

Type `,quit` or press Ctrl-D on an empty line to exit the REPL.

2.4 Running Script Files

While the REPL is great for exploration, real programs live in files. Create a file called `hello.scm`:

```
(import (scheme base) (scheme write))

(display "Hello, world!")
(newline)
```

Run it from the command line:

```
kaappi hello.scm
```

Output:

```
Hello, world!
```

Why the import line?. Kaappi pre-loads all of its built-in procedures, so this script would run even without the `import` line. Writing it anyway is a good habit: it documents which libraries your code depends on, and other R7RS implementations require it—they raise an "unbound variable" error for anything not imported. Script files in this book include `import` lines for that reason. Chapter 12 covers the library system in detail.

2.5 Editor Support

You can write Scheme in any text editor, but language server support makes the experience significantly better.

VS Code

Install the `kaappi` extension from the VS Code marketplace. It provides syntax highlighting, bracket matching, and full LSP integration (go-to-definition, find-references, document symbols) via the bundled `kaappi-lsp` server.

Neovim, Emacs, and Helix

Configure your editor's LSP client to use `kaappi-lsp` as the language server for `.scm` and `.sld` files. Note that the install script does not include `kaappi-lsp`: build it from source (`zig build` produces it at `zig-out/bin/kaappi-lsp` alongside the interpreter). The VS Code extension needs no extra step because it bundles its own copy.

Shell Completions

Enable tab completion for the `kaappi` command itself:

```
# For zsh (add to ~/.zshrc)
eval "$(kaappi --completions zsh)"

# For bash (add to ~/.bashrc)
eval "$(kaappi --completions bash)"

# For fish (add to config.fish)
kaappi --completions fish | source
```

> **Editor setup is optional.** For the purposes of this book, the REPL and a plain text editor are sufficient. Do not let editor configuration slow you down—you can always set up LSP later.

2.6 The Package Manager

Kaappi ships with `thottam`, a package manager for installing third-party libraries. You will not need it for the first several chapters, but here is a quick preview of how it works.

Install a library:

```
thottam install kaappi-json
```

This downloads the `kaappi-json` library and its dependencies to `~/.kaappi/lib/`. Once installed, you can import it in any Scheme file:

```
(import (kaappi json))

(define data (json-read (open-input-file "config.json")))
```

List installed packages:

```
thottam list
```

The ecosystem includes libraries for HTTP, web frameworks, PostgreSQL, SQLite, Redis, JSON, TOML, YAML, CSV, email, cryptography, logging, testing, and CLI tools. We will use several of these in later chapters; Appendix E lists them with brief descriptions.

Summary

- Install Kaappi with the one-line script, a manual binary download, or a source build, and confirm it with `kaappi --version`.
- The REPL provides multi-line input, line editing, tab completion, syntax highlighting, history, and the _ result variable.
- Run programs from files with `kaappi file.scm`; editors integrate through Kaappi's LSP server.
- The `thottam` package manager installs third-party libraries into `~/.kaappi/lib/`, which is always on the search path.

Exercises

Exercise 2.1: Explore the REPL

Launch the Kaappi REPL and experiment with its interactive features. Try each of the following:

1. Type `string-` and press Tab to see all string procedures. Pick one you have not used before and call it with an argument.
2. Define a variable, press Up to recall the definition, then edit it and press Enter to redefine it.
3. Enter an incomplete expression (e.g., type `(define (greet name)` and press Enter) to see the continuation prompt. Complete the function body on the next line.

Exercise 2.2: Your First Script File

Create a file called `greet.scm` with the following content, then run it with `kaappi greet.scm`:

```
(import (scheme base) (scheme write))

(define (greet name)
  (display "Hello, ")
```

```
  (display name)
  (display "!")
  (newline))

(greet "World")
(greet "Kaappi")
```

Verify that it prints two greeting lines:

```
Hello, World! Hello, Kaappi!
```

Then modify it to greet your own name.

Exercise 2.3: REPL Comma-Commands

The REPL provides special commands prefixed with a comma. Type `,help` at the REPL prompt to see the full list. Try each of the following and observe what happens:

```
> ,help
> ,env
> (define x 42)
> ,env
> ,quit
```

What does `,env` show before and after you define `x`? What other comma-commands look useful for debugging?

Appendix B is a complete reference for the command-line interface—commands, options, environment variables, and REPL comma-commands—when you need to look something up. In the next chapter, we will start writing Scheme in earnest.

Chapter 3
First Steps

With Kaappi installed and the REPL ready, it is time to write your first Scheme code. This chapter covers the fundamental building blocks: expressions, basic arithmetic, variables, and defining simple functions. Open a REPL and type along—every example here is meant to be tried interactively.

3.1 Everything Is an Expression

In Python, you distinguish between *statements* (which do things) and *expressions* (which produce values). In Scheme, there is only one category: everything is an expression, and every expression produces a value.

The syntax is uniform. An expression is either an atom (a number, string, or symbol standing alone) or a list enclosed in parentheses. When the evaluator sees a list, it treats the first element as the operator and the rest as operands:

```
> (+ 2 3)
5
> (* 6 7)
42
> (- 100 1)
99
```

The operator always comes first. This is called *prefix notation*. There is no infix `2 + 3` in Scheme—it is always `(+ 2 3)`.

No Precedence Rules

In Python, `2 + 3 * 4` evaluates to `14` because `*` binds tighter than `+`. In Scheme, you nest expressions explicitly:

```
> (+ 2 (* 3 4))
14
> (* (+ 1 2) (- 10 4))
18
```

The parentheses *are* the precedence. There is nothing to memorize, and there is never ambiguity about evaluation order.

Coming from Python. If `(+ 2 3)` looks strange, think of it as a procedure call where the procedure name goes inside the parentheses: like `add(2, 3)` but without the commas and with the procedure name moved inside. After a few examples, the pattern becomes natural.

Operators Take Multiple Arguments

Unlike most languages where + is binary, Scheme's arithmetic operators accept any number of arguments:

```
> (+ 1 2 3 4 5)
15
> (* 2 3 4)
24
> (+)
0
> (*)
1
```

With zero arguments, + returns 0 (the additive identity) and * returns 1 (the multiplicative identity).

Why prefix notation?. It is a deliberate trade, not an accident of history. Because the operator is named once, up front, operators can take any number of arguments—infix `1 + 2 + 3 + 4` names + three times to add four numbers. Because every expression has the same shape, there is no precedence table, and + is not special syntax but an ordinary procedure you can pass around like any other (Chapter 9 exploits this constantly). The uniform shape has one more consequence, and it is the big one: Scheme code is itself a Scheme data structure, which is what makes the macros of Chapter 11 possible.

3.2 Atoms

Atoms are the simplest values in Scheme—they cannot be broken into smaller parts.

Numbers

Kaappi supports a full numeric tower:

```
> 42
42
> -7
-7
> 3.14
3.14
> 1/3
1/3
> (+ 1/3 1/6)
1/2
```

Integers and exact rationals stay exact—there is no floating-point drift when you work with fractions. Kaappi also supports arbitrarily large integers (bignums):

```
> (* 1000000000 1000000000 1000000000)
1000000000000000000000000000
```

> **Coming from Python.** Python 3 also has arbitrary-precision integers, so this will feel familiar. The difference is that Scheme has exact rationals (`1/3`) as a built-in type—Python requires the `fractions` module.

Strings

Strings are delimited by double quotes:

```
> "hello, world"
"hello, world"
> (string-length "hello")
5
> (string-append "foo" "bar")
"foobar"
```

Single quotes mean something entirely different in Scheme (they denote quotation, which we will cover shortly), so strings are always double-quoted.

Booleans

The boolean values are `#t` (true) and `#f` (false). There is one critical rule that surprises Python programmers: **only `#f` is false**. Everything else—including `0`, the empty string, and the empty list—is true:

```
> (if 0 "truthy" "falsy")
"truthy"
> (if "" "truthy" "falsy")
"truthy"
> (if '() "truthy" "falsy")
"truthy"
> (if #f "truthy" "falsy")
"falsy"
```

> **Only `#f` is false.** This is one of the most common sources of bugs when coming from Python. In Scheme, `(if 0 ...)` takes the "then" branch because `0` is not `#f`. If you want to test whether a number is zero, use `(zero? n)` or `(= n 0)` explicitly.

> **Why so strict?.** Python's rule conflates "no result" with "a result that happens to be empty." Scheme's rule keeps them apart: a lookup procedure can return `0`, `""`, or the empty list as honest *data* and reserve `#f` for "not found," and the caller can tell the difference. Older Lisps went the other way—the empty list doubled as false—and Scheme deliberately separated the two. You will see this convention pay off throughout the standard library.

Characters

Individual characters are written with a `#\` prefix:

```
> #\a
#\a
> #\space
```

```
#\space
> #\newline
#\newline
> (char-alphabetic? #\Z)
#t
```

Characters are distinct from single-character strings.

Symbols

A symbol is a lightweight, interned name. Symbols are compared by identity (fast pointer comparison), not by character content like strings:

```
> 'hello
hello
> (symbol? 'hello)
#t
> (eq? 'abc 'abc)
#t
```

The `'` (single quote) is called *quote*. It prevents the expression that follows from being evaluated. Without it, `hello` would be looked up as a variable name, causing an "unbound variable" error. Quote works on entire lists too—`'(1 2 3)` builds a list of literal data without evaluating it. Chapter 5 covers quoting in depth.

3.3 Defining Variables

Use `define` to bind a name to a value:

```
> (define pi 3.14159)
> pi
3.14159
> (* pi 10 10)
314.159
```

Once defined, you can use the name anywhere an expression is expected. Names in Scheme can contain characters that would be illegal in most languages: hyphens, question marks, and exclamation marks are all common:

```
> (define my-favorite-number 42)
> my-favorite-number
42
```

Naming conventions. Scheme uses `kebab-case` (words separated by hyphens) instead of `snake_case` or `camelCase`. Predicates end with `?` (e.g., `zero?`, `string?`). Mutating procedures end with `!` (e.g., `set!`, `vector-set!`).

3.4 Defining Functions

Scheme officially calls these *procedures* rather than functions, because they can have side effects (like printing output). This book uses both terms interchangeably.

To define a procedure, put the name and parameters in a list after `define`:

```
> (define (square x) (* x x))
> (square 7)
49
> (square 1.5)
2.25
```

This is shorthand for the more explicit form using `lambda`, which creates an anonymous function:

```
> (define square (lambda (x) (* x x)))
> (square 5)
25
```

`lambda` is like Python's `lambda` keyword, but without the one-expression limitation—a Scheme lambda can contain any number of expressions in its body. For now, remember only that the two definition forms are equivalent; Chapter 7 explores `lambda` and anonymous functions fully.

Functions can take multiple parameters:

```
> (define (area width height) (* width height))
> (area 5 3)
15
```

And they can call other functions, including themselves:

```
> (define (sum-of-squares a b)
    (+ (square a) (square b)))
> (sum-of-squares 3 4)
```

```
25
```

Functions Are Values

Functions are ordinary values in Scheme. You can pass them as arguments to other functions:

```
> (define (apply-twice f x) (f (f x)))
> (apply-twice square 3)
81
> (apply-twice square 2)
16
```

In the first call, `(square 3)` gives `9`, then `(square 9)` gives `81`. In the second, `(square 2)` gives `4`, then `(square 4)` gives `16`. This ability to pass functions around is what makes Scheme a *functional* language, and it is the foundation for many powerful patterns you will learn in later chapters.

3.5 Basic Conditionals

The `if` expression has three parts: a test, a "then" expression, and an "else" expression:

```
> (if (> 5 3) "yes" "no")
"yes"
> (if (< 5 3) "yes" "no")
"no"
```

Because `if` is an expression (not a statement), it produces a value. You can use it anywhere a value is expected:

```
> (define (my-abs x)
    (if (< x 0) (- x) x))
> (my-abs -7)
7
> (my-abs 3)
3
```

(Scheme already provides this as the built-in `abs`; we define our own version to show the pattern. Had we called it `abs`, our definition would silently shadow the built-in one—Scheme lets you redefine anything.)

Coming from Python. This is like Python's ternary expression `"yes" if 5 > 3 else "no"`, but in Scheme the condition comes first. The pattern is always `(if test then else)`.

For multiple branches, use `cond`. It works like Python's `if/elif/else`:

```
> (define (classify n)
    (cond
      ((< n 0) "negative")
      ((= n 0) "zero")
      (else "positive")))
> (classify -5)
"negative"
> (classify 0)
"zero"
> (classify 42)
"positive"
```

Each clause is `(test result)`. The `else` clause matches when none of the earlier tests are true.

These two forms will carry you through the next few chapters. Chapter 6 treats conditionals properly—`cond`'s full clause syntax, `case`, `when`, `unless`, and the boolean combinators.

3.6 Comments

Line comments start with a semicolon and extend to the end of the line:

```
; This is a comment
(define x 10) ; inline comment
```

Block comments use `#|` and `|#`:

```
#|
This is a
multi-line comment
|#
```

A datum comment `#;` comments out the next expression:

```
> (+ 1 #;(this is ignored) 2)
3
```

This last form is particularly useful for temporarily disabling an argument or clause without deleting it.

3.7 Putting It Together

Let's write a slightly larger example that uses what we have learned. Here is a function that converts a temperature from Fahrenheit to Celsius:

```
> (define (fahrenheit→celsius f)
    (* (/ 5 9) (- f 32)))
> (fahrenheit→celsius 212)
100
> (fahrenheit→celsius 32)
0
> (fahrenheit→celsius 72)
200/9
```

Notice that `(fahrenheit→celsius 72)` returns the exact rational `200/9` rather than a floating-point approximation. Kaappi preserves exactness when all inputs are exact. If you want a decimal, use a floating-point literal:

```
> (define (fahrenheit→celsius-approx f)
    (* (/ 5.0 9.0) (- f 32)))
> (fahrenheit→celsius-approx 72)
22.2222222222222
```

Summary

- Scheme programs are built entirely from expressions in prefix notation, so there are no operator-precedence rules to memorize.
- The atomic data types are numbers, strings, booleans, characters, and symbols.
- `define` binds both variables and named functions, and functions are themselves first-class values.
- `if` and `cond` select between alternatives based on a condition.

- Code can be documented with line comments (`;`) and block comments.

Exercises

Exercise 3.1: Predict and Check

Before typing each expression into the REPL, predict what the result will be. Then check your answer:

```
> (+ (* 3 5) (- 10 6))
> (/ 1 3)
> (+ 1/4 1/4 1/4 1/4)
> (* (+ 1 2) (+ 3 4) (+ 5 6))
> (- (* 2 3) (* 4 5))
```

Which results surprised you? Pay attention to how Kaappi handles exact rationals versus integers.

Exercise 3.2: Celsius to Fahrenheit

The chapter showed a `fahrenheit→celsius` function. Write the inverse: a function `celsius→fahrenheit` that takes a temperature in Celsius and returns Fahrenheit. The formula is $F = C \times \frac{9}{5} + 32$. Test it in the REPL:

```
> (celsius→fahrenheit 0)
32
> (celsius→fahrenheit 100)
212
> (celsius→fahrenheit 37)
493/5
```

In Python, this would be `def celsius_to_fahrenheit(c): return c * 9/5 + 32`.

Exercise 3.3: Absolute Value

Write a function `(absolute-value n)` that returns the absolute value of `n` using **`if`**. It should handle positive numbers, negative numbers, and zero:

```
> (absolute-value -7)
7
> (absolute-value 5)
5
> (absolute-value 0)
0
```

Compare your solution to the `my-abs` function defined earlier in this chapter, and to Scheme's built-in `abs`. Are they equivalent?

Exercise 3.4: Recursive Power

Write a function `(power base exponent)` that computes $base^{exponent}$ using recursion. Assume `exponent` is a non-negative integer. The base case is: anything raised to the power 0 is 1. The recursive case is: $b^n = b \times b^{n-1}$.

```
> (power 2 10)
1024
> (power 3 0)
1
> (power 5 3)
125
```

In Python, you would write:

```
def power(base, exponent):
    if exponent == 0:
        return 1
    return base * power(base, exponent - 1)
```

Translate this logic into Scheme using **`if`**, `=`, and `-`. If recursion feels unfamiliar, lean on the Python version above and do not worry—Chapter 7 is devoted entirely to it.

Exercise 3.5: Temperature Classifier

Write a function `(describe-temp t)` that takes a temperature in Celsius and returns a string describing it. Use **cond** with these ranges:

- Below 0: `"freezing"`
- 0 to 15: `"cold"`
- 16 to 25: `"warm"`
- Above 25: `"hot"`

```
> (describe-temp -5)
"freezing"
> (describe-temp 10)
"cold"
> (describe-temp 22)
"warm"
> (describe-temp 35)
"hot"
```

Hint: you will need both $<$ and $\leqslant$ (or $\geqslant$) to express the boundary conditions.

In the next chapter, we will explore Scheme's data types in depth.

Part II

The Language

This is the heart of the book. These chapters cover the core of Scheme—data types, lists and pairs, control flow, functions, local bindings, higher-order functions, and input and output—introducing one concept at a time. Work through them in order, since each chapter builds directly on the ones before it.

Chapter 4
Data Types

Chapter 3 introduced the basic atoms in passing. This chapter examines Scheme's data types in depth: the full numeric tower, strings, characters, booleans, symbols, vectors, and bytevectors. You will learn how to create values of each type, the key procedures for working with them, and how to test what type a value is.

4.1 The Numeric Tower

Kaappi supports four kinds of numbers, organized in a hierarchy called the *numeric tower*:

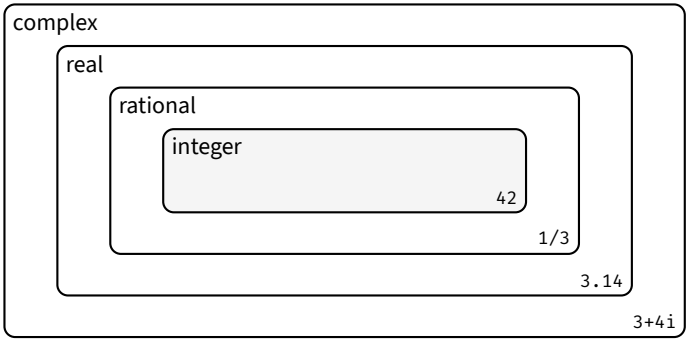
1. **Integers** — whole numbers, either fixnums (fast, 48-bit) or bignums (arbitrary precision)

2. **Rationals** — exact fractions like `1/3` or `-7/4`

3. **Real numbers** — flonums (IEEE 754 double-precision floating-point)

4. **Complex numbers** — pairs of reals, like `3+4i`

Every integer is also a rational (it is `n/1`). Every rational is also a real. Every real is also a complex (with zero imaginary part). The arithmetic procedures work uniformly across all levels.



Integers

Small integers (up to 48 bits) are stored as *fixnums*—unboxed values that require no heap allocation. When a fixnum overflows, Kaappi automatically promotes it to a *bignum* with unlimited precision:

```
> 42
42
> -100
-100
> (expt 2 100)
1267650600228229401496703205376
> (* 999999999999 999999999999)
999999999998000000000001
```

There is no silent wraparound. If the result does not fit in a fixnum, it becomes a bignum transparently.

Why 48 bits? A peek under the hood. Kaappi stores *every* Scheme value in a single 64-bit word using a technique called *NaN-boxing*: an IEEE 754 double has millions of bit patterns that all mean "not a number," and Kaappi hides its other types inside them. A flonum is stored as the double itself, so floating-point math never allocates memory; fixnums, characters, booleans, and pointers to heap objects live in the spare NaN patterns, which leaves 48 bits for a fixnum's payload. You never see any of this from Scheme—overflow promotes to bignums automatically—but it explains the 48, and it is why values cross the C boundary cheaply in Chapter 17.

Exact Rationals

When you divide two integers and the result is not an integer, Kaappi returns an exact rational:

```
> (/ 1 3)
1/3
> (/ 22 7)
22/7
> (+ 1/3 1/6)
1/2
> (* 3 1/3)
1
```

Rationals are always reduced to lowest terms. They stay exact through arbitrarily long chains of computation—no floating-point drift.

> **Coming from Python.** Python's `/` always returns a float. To get exact fractions in Python, you need `from fractions import Fraction`. In Scheme, exact arithmetic is the default.

Floating-Point Numbers

Floating-point numbers (flonums) are written with a decimal point or in scientific notation:

```
> 3.14
3.14
> 2.5e10
25000000000.0
> (/ 1.0 3.0)
0.3333333333333333
```

Exact vs. Inexact

Scheme distinguishes between *exact* and *inexact* numbers. Integers and rationals are exact; flonums are inexact. Mixing them produces an inexact result:

```
> (+ 1/3 0.1)
0.43333333333333335
> (exact? 1/3)
```

```
#t
> (inexact? 3.14)
#t
> (exact? 42)
#t
```

You can convert between exact and inexact representations:

```
> (inexact 1/3)
0.3333333333333333
> (exact 0.5)
1/2
```

Complex Numbers

Complex numbers are written in rectangular form with `+` or `-` separating the real and imaginary parts:

```
> 3+4i
3+4i
> (+ 1+2i 3+4i)
4.0+6.0i
> (magnitude 3+4i)
5.0
> (sqrt -1)
+i
```

Notice the results: Kaappi performs complex arithmetic on floating-point components, so complex results come back inexact even when the inputs are exact. (The empty real part in `+i` is shorthand for `0+1i`.)

Numeric Predicates and Comparisons

Scheme provides predicates to classify numbers and comparison procedures that work across the numeric tower:

```
> (number? 42)
#t
> (integer? 3.0)
#t
> (rational? 1/3)
#t
```

```
> (zero? 0)
#t
> (positive? 5)
#t
> (negative? -3)
#t
> (= 1 1.0)
#t
> (< 1 2 3 4 5)
#t
```

Note that <, >, ⩽, and ⩾ accept multiple arguments and check that the entire sequence is ordered.

Coming from Python. `(integer? 3.0)` returns `#t` because R7RS classifies numbers by their mathematical value, not their representation. The value 3.0 is mathematically an integer, even though it is stored as a flonum. In Python, `isinstance(3.0, int)` returns `False` because Python distinguishes by type, not by value.

Common Arithmetic Procedures

```
> (+ 1 2 3)
6
> (- 10 3)
7
> (* 2 3 4)
24
> (/ 10 3)
10/3
> (quotient 10 3)
3
> (remainder 10 3)
1
> (modulo 10 3)
1
> (remainder -10 3)
-1
> (modulo -10 3)
2
> (abs -7)
```

```
7
> (max 3 1 4 1 5)
5
> (min 3 1 4 1 5)
1
> (expt 2 10)
1024
> (sqrt 16)
4
```

`remainder` and `modulo` agree for positive operands but differ when signs are mixed: `remainder` takes the sign of the dividend, while `modulo` takes the sign of the divisor—the same behavior as Python's `%` operator.

4.2 Strings

Strings are sequences of Unicode characters, enclosed in double quotes. Kaappi stores strings as UTF-8 internally but indexes them by codepoint position:

```
> "hello, world"
"hello, world"
> (string-length "hello")
5
> (string-ref "hello" 0)
#\h
> (string-ref "hello" 4)
#\o
```

String Operations

```
> (string-append "foo" "bar" "baz")
"foobarbaz"
> (substring "hello, world" 7 12)
"world"
> (string-upcase "hello")
"HELLO"
> (string-downcase "HELLO")
"hello"
> (import (srfi 13))
> (string-contains "hello world" "world")
6
```

SRFI-13. `string-contains` comes from SRFI-13, the string library. In a script file, add `(import (srfi 13))` to use it. The other string procedures shown above are part of the R7RS standard and are always available.

String Conversion

```
> (number→string 42)
"42"
> (number→string 3.14)
"3.14"
> (string→number "123")
123
> (string→number "not-a-number")
#f
> (symbol→string 'hello)
"hello"
> (string→symbol "world")
world
```

`string→number` returns `#f` (not an error) if the string cannot be parsed as a number. This makes it safe to use on user input.

Mutating Strings

Strings in Scheme are mutable: `string-set!` replaces the character at an index in place:

```
> (define s (make-string 3 #\a))
> (string-set! s 0 #\Z)
> s
"Zaa"
```

There is one exception: string *literals* are immutable. Calling `string-set!` on a string that appears literally in your source code, like `(define s "abc")`, raises an error. Create fresh, mutable strings with `make-string` or `string-copy`.

In practice, idiomatic Scheme rarely mutates strings. Most programs build new strings from existing ones:

```
> (define greeting "hello")
```

```
> (string-append (string-upcase (substring greeting 0 1))
                 (substring greeting 1 5))
"Hello"
```

Coming from Python. Python strings are always immutable—operations like `upper()` return new strings. Scheme permits in-place mutation, but the functional style shown above is the norm; treat `string-set!` as a specialized tool, not a habit.

4.3 Characters

Chapter 3 introduced characters: single Unicode codepoints written with a `#\` prefix, such as `#\a`, `#\Z`, and the named forms `#\space`, `#\newline`, and `#\tab`. A character is a single value, distinct from a one-character string. What Chapter 3 did not cover are the predicates and conversions for working with them.

Character Predicates

```
> (char-alphabetic? #\a)
#t
> (char-numeric? #\5)
#t
> (char-whitespace? #\space)
#t
> (char-upper-case? #\A)
#t
> (char-lower-case? #\a)
#t
```

Character Conversion

```
> (char-upcase #\a)
#\A
> (char-downcase #\Z)
#\z
> (char→integer #\A)
65
```

```
> (integer→char 955)
#\λ
```

Characters and their codepoints round-trip. The last example produces the Greek lambda character—Kaappi has full Unicode support.

4.4 Booleans

Booleans were introduced in Chapter 3, but here is a summary of the key procedures:

```
> (boolean? #t)
#t
> (boolean? 0)
#f
> (not #f)
#t
> (not 0)
#f
> (not '())
#f
```

Remember: `not` returns `#t` only when given `#f`. Every other value—including `0`, the empty string, and the empty list—is considered true.

Boolean Combinators

`and` and `or` combine conditions; they short-circuit and return the determining value rather than a strict `#t` or `#f`. Chapter 6 covers them in depth alongside the other conditional forms.

4.5 Symbols

Symbols are interned identifiers. Two symbols with the same name are always the same object in memory, which makes comparison instantaneous:

```
> (eq? 'hello 'hello)
#t
> (eq? 'hello 'world)
#f
> (symbol? 'foo)
#t
```

```
> (symbol→string 'foo)
"foo"
> (string→symbol "bar")
bar
```

Symbols are commonly used as keys in association lists (alists), as tags in data structures, and as identifiers in programs that manipulate code.

```
> (define colors '((red . "#ff0000")
                   (green . "#00ff00")
                   (blue . "#0000ff")))
> (assq 'green colors)
(green . "#00ff00")
> (cdr (assq 'green colors))
"#00ff00"
```

The dotted `(red . "#ff0000")` pairs and the `cdr` accessor belong to Chapter 5; for now, read `(assq 'green colors)` as "look up the entry tagged `green`" and `cdr` as "take its value."

4.6 Vectors

Vectors are fixed-size, indexed collections. Indexing works like a Python list, but a vector cannot grow or shrink after creation—it is closer to a fixed-length array. They provide O(1) access by index:

```
> (define v (vector 10 20 30 40 50))
> (vector-ref v 0)
10
> (vector-ref v 4)
50
> (vector-length v)
5
```

Vector literals are written with `#(`:

```
> #(1 2 3)
#(1 2 3)
```

Unlike Scheme lists (which are linked), vectors support efficient random access. Use vectors when you need to look up elements by position.

Modifying Vectors

Vectors are mutable—you can change individual elements:

```
> (define v (vector 1 2 3))
> (vector-set! v 1 99)
> v
#(1 99 3)
```

Vector Operations

```
> (vector-map + #(1 2 3) #(10 20 30))
#(11 22 33)
> (vector→list #(a b c))
(a b c)
> (list→vector '(x y z))
#(x y z)
> (make-vector 5 0)
#(0 0 0 0 0)
```

4.7 Bytevectors

Bytevectors are sequences of bytes (unsigned 8-bit integers). They are used for binary data, file I/O, and interfacing with C libraries:

```
> (define bv #u8(72 101 108 108 111))
> (bytevector-length bv)
5
> (bytevector-u8-ref bv 0)
72
> (utf8→string bv)
"Hello"
```

You can create bytevectors from strings and vice versa:

```
> (string→utf8 "hello")
#u8(104 101 108 108 111)
> (utf8→string #u8(104 101 108 108 111))
"hello"
```

4.8 Type Predicates

Scheme provides predicates to test the type of any value. Each returns `#t` or `#f`:

```
> (number? 42)
#t
> (string? "hello")
#t
> (char? #\a)
#t
> (boolean? #t)
#t
> (symbol? 'foo)
#t
> (vector? #(1 2 3))
#t
> (bytevector? #u8(1 2 3))
#t
> (pair? '(1 2 3))
#t
> (null? '())
#t
> (procedure? +)
#t
```

These predicates are essential for writing functions that handle different types of input, and they come up frequently when processing heterogeneous data.

4.9 Equality: `eq?`, `eqv?`, `equal?`, and `=`

Scheme has three general-purpose equality predicates plus the numeric `=`. Python collapses most of this into `==` and `is`, so the distinctions deserve a careful look now—every chapter that follows relies on them.

`eq?` tests *identity*: are the two arguments the very same object in memory? It is the right choice for symbols and booleans, which are interned:

```
> (eq? 'apple 'apple)
#t
> (eq? '(1 2) '(1 2))
#f
> (eq? "hi" "hi")
#f
```

The two lists print the same but are separate objects, so `eq?` says `#f`. The same goes for strings.

`eqv?` extends `eq?` with well-defined behavior for numbers and characters. Two numbers are `eqv?` when they have the same value *and* the same exactness:

```
> (eqv? 42 42)
#t
> (eqv? 1.5 1.5)
#t
> (eqv? #\a #\a)
#t
> (eqv? 1 1.0)
#f
> (eqv? "hi" "hi")
#f
```

`equal?` compares *structure*, recursively. Two values are `equal?` when they look the same—this is the predicate that behaves like Python's `==` for containers:

```
> (equal? '(1 (2 3)) '(1 (2 3)))
#t
> (equal? "hello" "hello")
#t
> (equal? #(1 2) #(1 2))
#t
```

Finally, `=` compares *numeric value only* and works across exactness—but accepts nothing except numbers:

```
> (= 1 1.0)
#t
> (= 1/2 0.5)
#t
```

Predicate	Compares	Use for
`eq?`	object identity	symbols, booleans
`eqv?`	identity + value for numbers/chars	characters, exact numbers
`equal?`	structure, recursively	lists, strings, vectors
`=`	numeric value	numbers

When in doubt, `equal?` is the safe general-purpose choice; reach for the cheaper `eq?`/`eqv?` when identity is what you mean. For strings there is also the dedicated `string=?` predicate, and for characters `char=?`.

> **Coming from Python.** `eq?` corresponds to Python's `is`, and `equal?` to `==` on containers. Python has no direct analog of `eqv?`—think of it as `is` that also works sensibly for numbers and characters.

Summary

- Scheme's numeric tower spans integers, exact rationals, real (floating-point) numbers, and complex numbers, with a key distinction between exact and inexact values.
- Strings are sequences of Unicode characters (mutable, except literals—though idiomatic code builds new strings instead), and characters are a distinct type.
- Symbols are interned names, and the only booleans are `#t` and `#f`.
- Vectors are fixed-length, mutable, constant-time-indexed sequences; bytevectors hold raw bytes.
- Type predicates such as `number?`, `string?`, and `vector?` test a value's type.
- `eq?` tests identity, `eqv?` adds numbers and characters, `equal?` compares structure, and `=` compares numeric value.

Exercises

Exercise 4.1: Exploring the Numeric Tower

Create one value of each numeric type: an integer, a rational, a flonum, and a complex number. For each, use `exact?` and `inexact?` to confirm whether it is exact or inexact. Then convert between representations:

```
> (inexact 1/3)
???
> (exact 0.75)
```

```
???
> (exact 3+4i)
???
```

What do you get when you convert `0.1` to exact form with `(exact 0.1)`? Why is the result not `1/10`? (Hint: think about how floating-point numbers are stored.)

Exercise 4.2: Type Inspector

Write a function `type-name` that takes a single argument and returns a string describing its type. Use the type predicates from this chapter. Your function should distinguish at least: `"boolean"`, `"integer"`, `"rational"`, `"real"`, `"complex"`, `"string"`, `"char"`, `"symbol"`, `"vector"`, `"bytevector"`, `"pair"`, `"null"`, and `"procedure"`. Test the order of your checks carefully—remember that every integer is also rational, and every rational is also real.

```
> (type-name 42)
"integer"
> (type-name 1/3)
"rational"
> (type-name "hello")
"string"
> (type-name +)
"procedure"
```

Exercise 4.3: Counting Vowels

Write a function `count-vowels` that takes a string and returns the number of vowels (a, e, i, o, u—both upper and lower case) it contains. Start from this skeleton, which walks the string by index the same way the Recursive Power exercise from Chapter 3 counted down its exponent:

```
(define (count-vowels-from s i)
  (if (= i (string-length s))
```

```
      0
      (+ (if (vowel? (string-ref s i)) 1 0)
         (count-vowels-from s (+ i 1)))))

(define (count-vowels s)
  (count-vowels-from s 0))
```

Your job is to write the missing `vowel?` predicate. You will need `char-downcase` and a series of `char=?` checks combined with **`or`**. If the recursive skeleton does not click yet, revisit it after Chapter 7.

```
> (count-vowels "Hello, World!")
3
> (count-vowels "AEIOU")
5
> (count-vowels "rhythm")
0
```

In Python, you might write `sum(1 for c in s.lower() if c in "aeiou")`. How does your Scheme version compare?

Exercise 4.4: Vector of Squares

Create a vector containing the first 10 perfect squares (1, 4, 9, ..., 100) using `vector-map`. Start with a vector of `1` through `10` and map a squaring function over it:

```
> (define nums (vector 1 2 3 4 5 6 7 8 9 10))
> (vector-map ??? nums)
#(1 4 9 16 25 36 49 64 81 100)
```

Then try building the same vector a different way: use `make-vector` to create a 10-element vector, then use `vector-set!` in a recursive helper function to fill each position with `(* i i)`. (Save this second part for after Chapter 7 if recursion is new to you.)

The next chapter covers the most important compound data structure: lists and pairs.

λ

Chapter 5
Lists and Pairs

Lists are the most important data structure in Scheme. They are used for storing collections, building trees, representing structured data, and even encoding the source code of programs. Understanding lists is essential for everything that follows in this book.

Under the hood, lists are built from a simpler structure called a *pair* (also known as a *cons cell*). This chapter covers pairs, list construction, list access, quoting, common operations, and recursive list processing.

5.1 Pairs: The Building Block

A pair is a value that holds two things: a `car` (the first element) and a `cdr` (the second element). You create a pair with `cons`:

```
> (cons 1 2)
(1 . 2)
> (cons "hello" "world")
("hello" . "world")
```

The notation `(1 . 2)` is called a *dotted pair*. The dot separates the car from the cdr. You extract the two halves with `car` and `cdr`:

```
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```

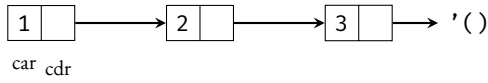
Why "car" and "cdr"?. These names come from the 1950s IBM 704 computer, where they stood for "Contents of the Address Register" and "Contents of the Decrement Register." They are historical artifacts, but every Scheme programmer knows them. Think of `car` as "first" and `cdr` as "rest."

5.2 Lists as Chains of Pairs

A list is a chain of pairs where each pair's cdr points to the next pair, and the last pair's cdr is the *empty list* `'()`:

```
> (cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```

Here is the structure visualized:



Each box is a pair. The left half is the car (the element), the right half is the cdr (a pointer to the next pair).

When the cdr chain ends with `'()`, we call it a *proper list*. Scheme hides the dots when printing proper lists—`(1 . (2 . (3 . ())))` is displayed as `(1 2 3)`.

5.3 Creating Lists

The `list` Procedure

The `list` procedure is the most convenient way to create a list:

```
> (list 1 2 3 4 5)
(1 2 3 4 5)
> (list "a" "b" "c")
("a" "b" "c")
> (list)
()
```

Quoting

For lists of literal values, you can use a quote:

```
> '(1 2 3)
(1 2 3)
> '(red green blue)
(red green blue)
> '(1 "two" #t)
(1 "two" #t)
```

The quote `'` prevents evaluation. Without it, `(red green blue)` would try to call `red` as a procedure. With the quote, it creates a list of three symbols.

The apostrophe is reader shorthand: `'x` is exactly the same as writing `(quote x)`. The reader expands one into the other before evaluation, which is why quoted data sometimes prints back with `quote` spelled out.

> **Quote vs. list.** `'(1 2 3)` and `(list 1 2 3)` both produce the list `(1 2 3)`, but they differ in an important way: quoted lists contain literal values only. If you need computed elements, use `list`:
>
> ```
> > (define x 10)
> > '(x 20 30)
> (x 20 30)
> > (list x 20 30)
> (10 20 30)
> ```
>
> In the quoted version, `x` is the symbol `x`, not the value 10.

Quasiquotation

Often you want a list that is *mostly* literal with a few computed elements. Rather than assembling it with `list` and `cons`, use *quasiquote*: a backtick `` ` `` quotes the whole structure, and a comma `,` (*unquote*) switches evaluation back on for one element:

```
> `(1 2 ,(+ 1 2))
(1 2 3)
> (define name "world")
> `("hello" ,name)
("hello" "world")
```

`,@` (*unquote-splicing*) evaluates to a list and splices its elements into place, rather than inserting the list as a single element:

```
> `(1 ,@(list 2 3) 4)
(1 2 3 4)
```

Quasiquote shines when building nested, tagged data:

```
> (define port 8080)
> `(config (host "localhost") (port ,port))
(config (host "localhost") (port 8080))
```

Like `'`, the backtick and comma are reader shorthand—for `quasiquote`, `unquote`, and `unquote-splicing`. Think of quasiquote as Scheme's template string: `` `(port ,port) `` plays the role of Python's `f"port {port}"`, except it produces structured data instead of text.

Building Lists with `cons`

The idiomatic way to build a list incrementally is to `cons` elements onto the front:

```
> (cons 1 '())
(1)
> (cons 1 '(2 3))
(1 2 3)
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
```

This is O(1)—adding an element to the front of a list is instantaneous because it just creates one new pair.

5.4 Accessing List Elements

`car` and `cdr`

```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
> (car (cdr '(a b c)))
b
```

```
> (cdr (cdr '(a b c)))
(c)
> (car (cdr (cdr '(a b c))))
c
```

`car` gives you the first element. `cdr` gives you the rest of the list (everything except the first element). Both are O(1).

Shorthand Compositions

For deeply nested access, Scheme provides shorthand procedures up to four levels deep:

```
> (cadr '(a b c))
b
> (caddr '(a b c))
c
> (caar '((1 2) (3 4)))
1
> (cadar '((1 2) (3 4)))
2
```

`cadr` is `(car (cdr ...))`, `caddr` is `(car (cdr (cdr ...)))`, and so on. Read the letters between `c` and `r` from right to left: each `a` means `car`, each `d` means `cdr`.

`list-ref`

For positional access by index:

```
> (list-ref '(a b c d e) 0)
a
> (list-ref '(a b c d e) 3)
d
```

Unlike vectors, `list-ref` is O(n)—it must walk the chain from the head. If you need frequent random access, use a vector instead.

5.5 Common List Operations

`length`

```
> (length '(a b c))
3
> (length '())
0
```

append

Concatenates lists:

```
> (append '(1 2) '(3 4) '(5 6))
(1 2 3 4 5 6)
> (append '(a b) '())
(a b)
```

`append` copies all lists except the last one. It is O(n) in the total length of all lists except the last.

reverse

```
> (reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

member and assoc

`member` searches a list for a value:

```
> (member 3 '(1 2 3 4 5))
(3 4 5)
> (member 9 '(1 2 3 4 5))
#f
```

It returns the tail of the list starting with the match, or `#f` if not found. This design lets you use the result as a boolean (since any non-`#f` value is true) while also giving you the position in the list.

`assoc` searches an association list (a list of pairs) by key:

```
> (define phonebook
    '(("Alice" . "555-1234")
      ("Bob" . "555-5678")
```

```
      ("Carol" . "555-9012")))
> (assoc "Bob" phonebook)
("Bob" . "555-5678")
> (cdr (assoc "Bob" phonebook))
"555-5678"
> (assoc "Dave" phonebook)
#f
```

Both procedures come in three variants that differ only in the equality predicate they use (Section 4.9): `member` and `assoc` compare with `equal?`, `memq` and `assq` with `eq?`, and `memv` and `assv` with `eqv?`. Reach for the `q` variants when the elements or keys are symbols, and the `v` variants for numbers and characters—both skip the structural comparison that `equal?` performs:

```
> (memq 'c '(a b c d))
(c d)
> (memv 3 '(1 2 3 4))
(3 4)
> (assq 'green '((red . "#ff0000") (green . "#00ff00")))
(green . "#00ff00")
```

Coming from Python. An association list is like a Python dictionary, but with O(n) lookup. For small collections (under 50 entries), alists are simple and effective. For larger collections, use hash tables (covered in Chapter 16).

5.6 List Predicates

```
> (pair? '(1 2 3))
#t
> (pair? '())
#f
> (null? '())
#t
> (null? '(1 2 3))
#f
> (list? '(1 2 3))
#t
> (list? (cons 1 2))
#f
```

`pair?` tests whether something is a cons cell. `null?` tests for the empty list. `list?` tests for a proper list (a chain ending in `'()`). Note that `(list? (cons 1 2))` returns `#f` because the cdr is `2`—not a pair or `'()`—so the chain does not terminate properly.

5.7 Nested Lists and Trees

Lists can contain other lists, forming trees:

```
> '((1 2) (3 4) (5 6))
((1 2) (3 4) (5 6))
> (car '((1 2) (3 4)))
(1 2)
> (caar '((1 2) (3 4)))
1
```

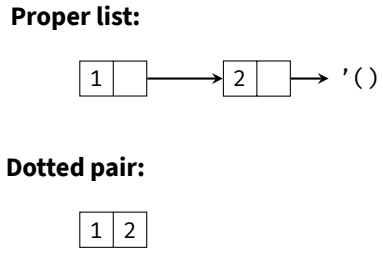
Processing nested lists to arbitrary depth requires recursion, which Chapter 7 introduces—we return to tree-shaped data there.

5.8 Improper Lists and Dotted Pairs

Not all pairs form proper lists. A pair whose cdr is not a list or the empty list is called an *improper list* or *dotted pair*:

```
> (cons 1 2)
(1 . 2)
> (cons 1 (cons 2 3))
(1 2 . 3)
```

Compare the two structures:



Dotted pairs are useful as lightweight two-element structures—association lists use them for key-value entries. But most list-processing procedures (`length`, `map`, `for-each`) expect proper lists and will error on improper ones.

Mutating pairs. Pairs are mutable: `set-car!` and `set-cdr!` replace one half of an existing pair in place.

```
> (define p (cons 1 2))
> (set-car! p 99)
> p
(99 . 2)
```

Idiomatic Scheme rarely uses them—building a new list is usually clearer and safer, as in the alist-update idiom in the exercises below. One hard rule: never mutate a quoted literal such as `'(1 2 3)`; R7RS declares that an error. If you intend to mutate, construct the pairs with `cons` or `list`.

Shared Structure and Datum Labels. Because pairs are references, the same pair can appear in a structure more than once. R7RS has reader syntax for writing such sharing literally: `#0=` labels the datum that follows, and `#0#` refers back to it (any number works, not just 0):

```
> (define shared '(#0=(a b) #0#))
> shared
((a b) (a b))
> (eq? (car shared) (cadr shared))
#t
```

Plain `write` prints the two elements as if they were separate lists; `write-shared` makes the sharing visible using the same notation:

```
> (write-shared shared)
(#0=(a b) #0#)
```

Labels are also how `write` prints *circular* data—a structure that contains itself, such as `#0=(1 2 . #0#)`—without looping forever. You will

rarely type datum labels yourself, but you will meet them in output whenever a structure contains a cycle.

5.9 Performance Characteristics

Understanding list performance helps you choose the right data structure:

Operation	Complexity	Notes
`cons`	O(1)	Allocates one pair
`car`, `cdr`	O(1)	Pointer access
`length`	O(n)	Must traverse entire list
`list-ref`	O(n)	Walks from the head
`append`	O(n)	Copies first argument(s)
`reverse`	O(n)	Creates a new list
`member`, `assoc`	O(n)	Linear search

Lists excel at sequential processing and stack-like access (always working with the front). When you need O(1) random access, use vectors. When you need O(1) key lookup, use hash tables.

Summary

- A pair (cons cell) holds two values—its `car` and `cdr`—and lists are chains of pairs ending in the empty list.
- Build lists with `list`, `cons`, or quoting, and access them with `car`, `cdr`, and `list-ref`.
- Common operations include `length`, `append`, `reverse`, and the `member`/`assoc` lookup families (`memq`, `memv`, `assq`, `assv`).
- Lists have a recursive structure—empty, or an element plus a smaller list—which Chapter 7 exploits to process them.
- Lists offer constant-time prepend but linear-time access and length, which guides when to use them.

Exercises

Exercise 5.1: Building Lists by Hand

Construct the list `(a b c d)` using only `cons` and `'()`—do not use `list` or quoting. Then verify that your result is equal to `(list 'a 'b 'c 'd)` using `equal?`. Draw the pair diagram on paper to reinforce how the chain of cons cells forms a proper list.

Exercise 5.2: Pair Anatomy

Predict the result of each expression, then check your answers in the REPL:

```
(car '((1 2) (3 4)))
(cdr '((1 2) (3 4)))
(caar '((1 2) (3 4)))
(cadr '((1 2) (3 4)))
(cons '(a) '(b c))
(append '(a) '(b c))
```

Pay attention to the last two: why do `cons` and `append` give different results for the same arguments?

Exercise 5.3: A Phone Book with `assoc`

Define an association list `phone-book` mapping names (strings) to numbers. Then:

1. Look up an existing name with `assoc`. What does it return?
2. Look up a missing name. What do you get instead?
3. "Add" an entry by consing a new pair onto the front, and look it up.
4. Cons a second entry with an existing name onto the front. Which entry does `assoc` find, and why?

The behavior in step 4—new entries shadow old ones—is a common functional idiom for updating alists without mutation.

Exercise 5.4: List Surgery

Using only `append`, `reverse`, `list`, `car`, `cdr`, `length`, and `list-ref`—no recursion—write:

1. `rotate-left`: move the first element to the end.
2. `last-element`: return the final element of a non-empty list.

```
> (rotate-left '(1 2 3 4))
(2 3 4 1)
> (last-element '(a b c))
c
```

Both have one-line solutions. What is the time complexity of each?

In the next chapter, we look at control flow—conditionals, boolean logic, sequencing, and loops.

Chapter 6
Control Flow

In most languages, control flow is managed with statements: `if/else` blocks, `for` loops, `switch` statements. In Scheme, all control flow is handled by expressions that produce values. This chapter covers `if`, `cond`, `case`, boolean combinators, `when`/`unless`, sequencing with `begin`, and iteration with `do`.

6.1 `if`: The Fundamental Conditional

The `if` expression is the most basic conditional. It has the form:

```
(if test consequent alternate)
```

If `test` evaluates to any true value (anything that is not `#f`), the `consequent` is evaluated and its value returned. Otherwise, `alternate` is evaluated and returned:

```
> (if (> 3 2) "yes" "no")
"yes"
> (if (< 3 2) "yes" "no")
"no"
> (if (zero? 0) 'zero 'nonzero)
zero
```

Because `if` is an expression, you can use it anywhere a value is expected:

```
> (+ 1 (if #t 10 20))
11
> (string-append "The answer is "
                 (if (even? 4) "even" "odd"))
"The answer is even"
```

One-Armed `if`

You can omit the alternate:

```
> (if (> 5 3) (display "big\n"))
big
```

When the test is false and there is no alternate, the result is unspecified. In practice, use `when` (see below) for one-armed conditionals—it communicates intent more clearly.

6.2 cond: Multiple Branches

`cond` is Scheme's equivalent of Python's `if/elif/else`. It tests multiple conditions in order:

```
> (define (describe-temp celsius)
    (cond
      ((< celsius 0) "freezing")
      ((< celsius 10) "cold")
      ((< celsius 20) "mild")
      ((< celsius 30) "warm")
      (else "hot")))
> (describe-temp -5)
"freezing"
> (describe-temp 15)
"mild"
> (describe-temp 35)
"hot"
```

Each clause is `(test expr ...)`. The first clause whose test returns true has its expressions evaluated, and the last one's value is returned. The `else` clause matches if nothing else did. If no test succeeds and there is no `else`, the result is unspecified—when you use `cond` for its value, always include an `else` clause.

The ⇒ Syntax

A special clause form `(test => proc)` passes the test result to a procedure:

```
> (cond
    ((assv 2 '((1 "one") (2 "two") (3 "three")))
     => cadr)
    (else "not found"))
```

```
"two"
```

Here, `assv` returns `(2 "two")` (a true value), and ⇒ passes it to `cadr`, which extracts `"two"`. This avoids evaluating the lookup twice.

6.3 case: Matching Against Values

`case` compares a value against sets of constants, similar to a `switch` statement in C or `match` on literals in Python:

```
> (define (day-type day)
    (case day
      ((monday tuesday wednesday thursday friday)
       "weekday")
      ((saturday sunday)
       "weekend")
      (else "unknown")))
> (day-type 'saturday)
"weekend"
> (day-type 'tuesday)
"weekday"
```

Each clause lists datum values in parentheses. The key is compared using `eqv?` (Section 4.9), which tests equality for numbers, characters, booleans, and symbols (but not strings—two strings with the same content may not be `eqv?`). For matching against strings, use `cond` with `string=?` instead. As with `cond`, if no clause matches and there is no `else`, the result is unspecified.

```
> (case (+ 2 3)
    ((1 2 3) "small")
    ((4 5 6) "medium")
    (else "large"))
"medium"
```

Coming from Python. `case` is similar to Python 3.10's `match/case` but limited to literal value matching (no pattern destructuring). For pattern matching on structure, Scheme uses recursive functions or `cond` with predicates.

6.4 Boolean Combinators: `and` and `or`

`and` and `or` are short-circuiting expressions that return the determining value—not just `#t` or `#f`.

`and`

`and` evaluates its arguments left to right. If any is `#f`, it returns `#f` immediately. If all are true, it returns the last value:

```
> (and 1 2 3)
3
> (and 1 #f 3)
#f
> (and (> 3 2) (< 5 10) "both true")
"both true"
> (and)
#t
```

`or`

`or` evaluates its arguments left to right. If any is true, it returns that value immediately. If all are `#f`, it returns `#f`:

```
> (or #f #f 42)
42
> (or #f #f #f)
#f
> (or (memq 'b '(a b c)) 'not-found)
(b c)
> (or)
#f
```

Using `or` for Defaults

A common pattern is using `or` to provide a fallback when a value may be `#f`:

```
> (define (server-port configured)
    (or configured 8080))
> (server-port 3000)
3000
> (server-port #f)
```

```
8080
```

Because searching procedures such as `assoc` and `member` return `#f` on failure, their results chain naturally into `or`. Note that `or` falls through only on `#f`—unlike in Python, `0` and `""` are true values and will not trigger the default.

Using and for Guarded Access

`and` is useful for guarding an operation that would error on the wrong input—the guard runs first, and the operation runs only if it passes:

```
> (define (safe-first lst)
    (and (pair? lst) (car lst)))
> (safe-first '(1 2 3))
1
> (safe-first '())
#f
```

If `lst` is empty, `(car lst)` would raise an error; the `and` short-circuits to `#f` instead. For the lookup-then-extract pattern—testing an `assoc` result and then taking its `cdr`—prefer the ⇒ clause of `cond` shown earlier, which avoids repeating the lookup.

> **Coming from Python.** Like Python's `and`/`or`, these return the deciding operand rather than a strict boolean, so `(and x (f x))` works like Python's `x and f(x)`. The crucial difference is what counts as false: Python treats `0`, `""`, and `[]` as falsy, but Scheme has exactly one false value, `#f`. `(or 0 42)` is `0`, not `42`—port idioms like `val or default` with care.

6.5 not

`not` negates a boolean value. It returns `#t` only when given `#f`:

```
> (not #f)
#t
> (not #t)
#f
> (not 0)
#f
> (not '())
```

```
#f
```

> **Zero is true in Scheme.** Since only `#f` is false in Scheme, `(not 0)` is `#f`—not `#t`. This is the opposite of Python, where `not 0` is `True`. The same applies to the empty list, the empty string, and every other value: only `#f` is false.

6.6 `when` and `unless`

`when` is a one-armed conditional for side effects—it executes its body only if the test is true:

```
> (define x 10)
> (when (> x 5)
    (display "x is large")
    (newline))
x is large
```

`unless` is the opposite—it executes when the test is false:

```
> (unless (= x 0)
    (display "x is nonzero")
    (newline))
x is nonzero
```

Both forms accept multiple expressions in their body (implicit `begin`). Use them instead of one-armed `if` when you want to perform actions without needing a result value.

6.7 `begin`: Sequencing

`begin` groups multiple expressions into one, evaluating them in order and returning the last value:

```
> (begin
    (display "first\n")
    (display "second\n")
    42)
first
```

```
second
42
```

You rarely need explicit `begin` because most forms (`define` function bodies, `when`, `unless`, `cond` clauses) already accept multiple body expressions. It is mainly useful inside `if` when you need multiple expressions in one branch:

```
> (if (> 5 3)
      (begin
        (display "yes\n")
        "positive")
      "negative")
yes
"positive"
```

6.8 do: Iterative Loops

Scheme has no dedicated `while` or `for` statement. Iteration is usually expressed with recursion, which Chapter 7 covers in depth. For loops with explicit loop variables, Scheme provides the `do` form:

```
(do ((var init step) ...)
    (test result ...)
  body ...)
```

Each variable starts at `init` and is updated to `step` each iteration. The loop terminates when `test` becomes true, returning `result`:

```
> (do ((i 0 (+ i 1))
       (sum 0 (+ sum i)))
      ((= i 10) sum))
45
```

Here is `do` building a reversed list:

```
> (do ((lst '(1 2 3 4 5) (cdr lst))
       (result '() (cons (car lst) result)))
      ((null? lst) result))
(5 4 3 2 1)
```

Coming from Python. `do` is roughly equivalent to a Python `for` or `while` loop. Most Scheme programmers prefer recursion (Chapter 7) or named `let` (Chapter 8) for loops—but `do` needs nothing beyond this chapter, and it shines when several loop variables change together.

6.9 Putting It All Together

Here is a function that demonstrates several control flow forms working together:

```
(define (fizzbuzz n)
  (do ((i 1 (+ i 1)))
      ((> i n))
    (cond
      ((and (zero? (modulo i 3))
            (zero? (modulo i 5)))
       (display "FizzBuzz"))
      ((zero? (modulo i 3))
       (display "Fizz"))
      ((zero? (modulo i 5))
       (display "Buzz"))
      (else
       (display i)))
    (display " ")))
```

```
> (fizzbuzz 15)
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz
```

This uses `do` for the loop, `cond` for the multi-branch logic, and `and` to combine conditions.

Summary

- `if` is the fundamental two-way conditional; `cond` and `case` handle multiple branches.
- `and` and `or` short-circuit and return the determining value, which is handy for defaults and guards.
- `when` and `unless` run a body under a single condition with no else branch.

- `begin` sequences expressions, and `do` expresses iterative loops.
- Because conditionals are expressions, they produce values rather than merely directing control.

Exercises

Exercise 6.1: Configurable FizzBuzz

The chapter showed a standard FizzBuzz implementation. Generalize it: write a function `fizzbuzz-custom` that takes an upper bound `n` and an association list of `(divisor . word)` pairs. For each number from 1 to `n`, print the concatenation of all matching words, or the number itself if no divisor matches:

```
> (fizzbuzz-custom 15 '((3 . "Fizz") (5 . "Buzz")))
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz
> (fizzbuzz-custom 10 '((2 . "Even") (3 . "Tri")))
1 Even Tri Even 5 EvenTri 7 Even Tri Even
```

Use **`do`** for the outer loop and a helper function to build the output for one number. This exercises **`cond`**, **`when`**, **`do`**, and string operations together.

Exercise 6.2: Letter Grades

Write a function `grade` that takes a numeric score (0–100) and returns a letter grade as a string: `"A"` for 90+, `"B"` for 80–89, `"C"` for 70–79, `"D"` for 60–69, and `"F"` below 60. Use **`cond`**:

```
> (grade 95)
"A"
> (grade 73)
"C"
> (grade 42)
"F"
```

Bonus: add input validation—return `"Invalid"` if the score is not between 0 and 100.

Exercise 6.3: Clamping a Value

Write a function `clamp` that takes three arguments—`value`, `lo`, and `hi`—and returns `value` restricted to the range [`lo`, `hi`]. Use **`cond`**:

```
> (clamp 5 1 10)
5
> (clamp -3 0 100)
0
> (clamp 150 0 100)
100
```

In Python, this is `max(lo, min(hi, value))`. How does the **`cond`** version compare for readability?

In the next chapter, we turn to functions in depth—including recursion, the technique Scheme programmers reach for where other languages use loops.

———— λ ————

Chapter 7
Functions

Functions are the primary organizational unit in Scheme. There are no classes and no methods—programs are built by defining and composing functions, with libraries (Chapter 12) grouping them into reusable units. This chapter covers defining functions, recursion, closures, tail-call optimization, variadic arguments and `case-lambda`, and returning multiple values.

7.1 Defining Functions

You have seen the shorthand form for defining functions:

```
> (define (square x) (* x x))
> (square 7)
49
```

This is syntactic sugar for binding a name to a `lambda` expression:

```
> (define square (lambda (x) (* x x)))
> (square 7)
49
```

Both forms are equivalent. The shorthand `(define (name params) body)` is more common in practice, but understanding the `lambda` form is essential because anonymous functions appear everywhere in Scheme.

Multiple Expressions in the Body

Unlike Python's `lambda` (limited to one expression), a Scheme lambda body can contain multiple expressions. The value of the last expression is returned:

```
> (define (greet name)
    (display "Hello, ")
    (display name)
    (newline)
    (string-append "Greeted " name))
> (greet "Alice")
Hello, Alice
"Greeted Alice"
```

The first three expressions produce side effects (printing). The last expression's value is the function's return value.

Functions with Multiple Parameters

```
> (define (hypotenuse a b)
    (sqrt (+ (* a a) (* b b))))
> (hypotenuse 3 4)
5
```

7.2 Anonymous Functions

You do not need to name every function. `lambda` creates a function value directly:

```
> ((lambda (x) (* x x)) 5)
25
> (map (lambda (x) (+ x 10)) '(1 2 3 4))
(11 12 13 14)
```

Anonymous functions are particularly useful when passing short operations to higher-order functions like `map` and `filter`. You will use them constantly.

> **Coming from Python.** `(lambda (x) (* x x))` is Scheme's equivalent of Python's `lambda x: x * x`. The difference is that Scheme's version can contain multiple expressions and arbitrary complexity—there is no need for a separate `def` just because the function has more than one line.

7.3 Recursion

Scheme has no dedicated `for` or `while` statement—aside from the `do` form from Chapter 6, repetition is expressed through recursion, and the language is designed to make recursion efficient and natural.

Simple Recursion

The classic factorial:

```
> (define (factorial n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
> (factorial 5)
120
> (factorial 20)
2432902008176640000
```

The structure is always the same: a base case that returns a known value, and a recursive case that reduces the problem.

Here is a function that computes the nth Fibonacci number:

```
> (define (fib n)
    (cond
      ((= n 0) 0)
      ((= n 1) 1)
      (else (+ (fib (- n 1))
               (fib (- n 2))))))
> (fib 10)
55
```

> **Exponential recursion.** This naive Fibonacci is $O(2^n)$—it recomputes the same subproblems repeatedly. For large `n`, use the tail-recursive version shown later in this chapter.

Recursion on Lists

Chapter 5 showed that a list is either empty or an element followed by a smaller list. That structure dictates the standard recursive pattern: check for empty (base case), process the first element, recurse on the rest. Counting elements:

```
(define (my-length lst)
  (if (null? lst)
      0
      (+ 1 (my-length (cdr lst)))))
```

```
> (my-length '(a b c d))
4
```

Summing a list of numbers:

```
(define (sum-list lst)
  (if (null? lst)
      0
      (+ (car lst) (sum-list (cdr lst)))))
```

```
> (sum-list '(1 2 3 4 5))
15
```

The same shape transforms a list into a new one:

```
> (define (my-map f lst)
    (if (null? lst)
        '()
        (cons (f (car lst))
              (my-map f (cdr lst)))))
> (my-map square '(1 2 3 4 5))  ; square was defined above
(1 4 9 16 25)
```

Coming from Python. This recursive pattern replaces Python's `for` loop. Instead of iterating with a counter, you decompose the list into head and tail. This style takes practice, but it eliminates off-by-one errors and makes the flow of data explicit.

Recursion on Nested Lists

When lists nest, the recursion branches: if the first element is itself a pair, recurse into it as well as into the rest. Here is a function that flattens a nested list into a single level:

```
(define (flatten lst)
  (cond
    ((null? lst) '())
    ((pair? (car lst))
     (append (flatten (car lst))
             (flatten (cdr lst))))
    (else
     (cons (car lst) (flatten (cdr lst))))))
```

```
> (flatten '(1 (2 3) (4 (5 6)) 7))
(1 2 3 4 5 6 7)
```

7.4 Tail Calls and Tail Recursion

When a recursive call is the *last thing* a function does—the final expression with nothing left to do after it returns—it is in *tail position*. Kaappi (and all conforming Schemes) guarantee that tail calls reuse the current stack frame instead of allocating a new one.

This means tail-recursive functions use constant stack space, just like a loop.

Non-Tail vs. Tail Recursive

The factorial above is *not* tail-recursive because after the recursive call, it still needs to multiply:

```
;; NOT tail-recursive: (* n ...) wraps the recursive call
(define (factorial n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

Here is the tail-recursive version using an accumulator:

```
> (define (factorial n)
    (define (loop i acc)
```

```
      (if (= i 0)
          acc
          (loop (- i 1) (* acc i))))
    (loop n 1))
> (factorial 20)
2432902008176640000
```

The call to `loop` is the last expression—nothing wraps it—so it runs without growing the stack, even for `(factorial 1000000)`. (The inner `define` creates a helper procedure visible only inside `factorial`; these *internal definitions* are covered in Chapter 8.)

Named `let` for Loops

Named `let` is the idiomatic way to write a tail-recursive loop. It defines a local function and calls it immediately:

```
> (let loop ((i 1))
    (when (⩽ i 5)
      (display i)
      (display " ")
      (loop (+ i 1))))
1 2 3 4 5
```

`loop` is a name you choose—it is bound to a function with parameter `i`. Calling `(loop (+ i 1))` recurses with the next value. Because the call is in tail position, it runs in constant space. Chapter 8 returns to named `let` and shows how it relates to the other `let` forms.

Here is a tail-recursive Fibonacci using named `let`:

```
> (define (fib n)
    (let loop ((i n) (a 0) (b 1))
      (if (= i 0)
          a
          (loop (- i 1) b (+ a b)))))
> (fib 50)
12586269025
```

This version is O(n), computing `(fib 50)` instantly.

Coming from Python. Named `let` is Scheme's equivalent of a `while` loop with loop variables. The "loop variables" are the parameters, and each "iteration" is a tail call with updated values. Once you internalize this pattern, you will find it as natural as `for i in range( ... )`.

7.5 Closures

A *closure* is a function that remembers the environment where it was created. Variables from the enclosing scope remain accessible even after that scope has exited.

```
> (define (make-adder n)
    (lambda (x) (+ x n)))
> (define add5 (make-adder 5))
> (define add10 (make-adder 10))
> (add5 3)
8
> (add10 3)
13
```

Each call to `make-adder` creates a new closure that captures its own copy of `n`. The two closures `add5` and `add10` are independent.

Closures with Mutable State

Closures can capture mutable variables, creating private state. The `(let ((n 0)) ... )` form below introduces a local variable `n` bound to 0—Chapter 8 covers `let` fully; for now, read it as "create a variable `n` visible only inside this body":

```
> (define (make-counter)
    (let ((n 0))
      (lambda ()
        (set! n (+ n 1))
        n)))
> (define c1 (make-counter))
> (define c2 (make-counter))
> (c1)
1
> (c1)
2
> (c1)
3
```

```
> (c2)
1
```

Each counter has its own private `n`. This is how you encapsulate state in Scheme without classes.

> **Coming from Python.** This is the same concept as Python closures. The difference is that Scheme uses closures as a primary organizational tool (where Python might reach for a class), and `set!` makes the captured variable mutable.

7.6 Variadic Functions

Functions can accept a variable number of arguments using *dot notation* in the parameter list:

```
> (define (sum . numbers)
    (let loop ((lst numbers) (acc 0))
      (if (null? lst)
          acc
          (loop (cdr lst) (+ acc (car lst))))))
> (sum 1 2 3)
6
> (sum 1 2 3 4 5)
15
```

The dot before `numbers` means "collect all remaining arguments into a list." You can also have required parameters before the rest:

```
> (define (format-message prefix . parts)
    (apply string-append prefix ": " parts))
> (format-message "ERROR" "file not found")
"ERROR: file not found"
```

7.7 `case-lambda`: Optional Arguments

Python expresses optional arguments with default values: `def greet(name, greeting="Hello")`. Scheme's answer is `case-lambda`, which builds a proce-

dure from several clauses, one per parameter list. When the procedure is called, the clause matching the number of arguments runs:

```
> (define greet
    (case-lambda
      ((name) (greet name "Hello"))
      ((name greeting)
       (string-append greeting ", " name "!"))))
> (greet "Alice")
"Hello, Alice!"
> (greet "Alice" "Howdy")
"Howdy, Alice!"
```

The one-argument clause supplies the default by calling the two-argument clause. A clause's parameter list can also use dot notation—`((x . rest) ...)`—to catch every remaining arity, just as in a variadic `define`.

7.8 apply

`apply` calls a function with arguments from a list:

```
> (apply + '(1 2 3 4 5))
15
> (apply max '(3 7 2 9 4))
9
> (apply string-append '("hello" " " "world"))
"hello world"
```

This is essential when you have a list of arguments and need to pass them to a function that expects individual parameters.

7.9 Multiple Return Values

Scheme functions can return multiple values using `values`:

```
> (define (divide a b)
    (values (quotient a b) (remainder a b)))
```

The caller must explicitly receive the values. The fundamental receiver is `call-with-values`, which hands them to another function as its arguments:

```
> (call-with-values (lambda () (divide 17 5)) list)
(3 2)
```

In practice you will usually reach for the more convenient `let-values` (covered in detail in Chapter 8), which binds each value to a name:

```
> (let-values (((q r) (divide 17 5)))
    (display "Quotient: ") (display q) (newline)
    (display "Remainder: ") (display r) (newline))
Quotient: 3
Remainder: 2
```

Coming from Python. This is similar to Python's tuple unpacking: `q, r = divmod(17, 5)`. Scheme's version is slightly more verbose but achieves the same result.

7.10 Function Composition Patterns

A few patterns come up frequently when working with functions.

Functions that Return Functions

```
> (define (compose f g)
    (lambda (x) (f (g x))))
> (define inc-then-square (compose square (lambda (x) (+ x
    1))))
> (inc-then-square 4)
25
```

Partial Application

```
> (define (partial f . fixed-args)
    (lambda rest-args
      (apply f (append fixed-args rest-args))))
> (define double (partial * 2))
> (double 5)
10
```

```
> (define add1 (partial + 1))
> (add1 99)
100
```

These patterns—composition, partial application, and higher-order functions in general—are the tools Scheme gives you in place of classes and inheritance. They are simpler, more composable, and often more powerful.

Summary

- Define named functions with `define` and anonymous ones with `lambda`; both create the same kind of procedure.
- Recursion is Scheme's primary looping mechanism, and tail-recursive calls run in constant space.
- Named `let` expresses loops as local tail recursion.
- Closures capture their defining environment, enabling private, mutable state.
- Variadic functions use a rest parameter, `case-lambda` dispatches on argument count (Scheme's optional arguments), and `apply` spreads a list into individual arguments.
- Procedures can return several results with `values`, received with `call-with-values` or `let-values`.

Exercises

Exercise 7.1: Tail-Recursive Summation

Write a function `sum-to` that computes the sum $1 + 2 + \cdots + n$ using tail recursion with an accumulator. Verify that it handles large values without stack overflow:

```
> (sum-to 10)
55
> (sum-to 1000000)
500000500000
```

Compare with the formula $n(n+1)/2$ to check correctness.

Exercise 7.2: Accumulator Closure

Write a function `make-accumulator` that returns a closure. Each time the closure is called with a number, it adds that number to a running total and returns the new total:

```
> (define acc (make-accumulator))
> (acc 10)
10
> (acc 25)
35
> (acc 5)
40
```

In Python, you would use a class with `__call__` or a closure over a `nonlocal` variable. In Scheme, `set!` and a **`let`**-bound variable do the job.

Exercise 7.3: Left-to-Right Composition

The chapter defined `compose`, which applies the *rightmost* function first: `((compose f g) x)` computes `(f (g x))`. Many pipelines read more naturally the other way around. Write `pipe`, which applies its functions left to right—`((pipe f g) x)` computes `(g (f x))`:

```
> (define inc-then-double
    (pipe (lambda (x) (+ x 1))
          (lambda (x) (* x 2))))
> (inc-then-double 3)
8
> (define exclaim
    (pipe (lambda (s) (string-append s "!"))
          string-upcase))
> (exclaim "hello")
"HELLO!"
```

Can you define `pipe` in terms of `compose`? Bonus (after reading Chapter 9): extend `pipe` to accept any number of functions using rest arguments and `fold`.

Exercise 7.4: Variadic Maximum

Write a function `my-max` that takes one or more numeric arguments and returns the largest. Use a required first parameter plus rest arguments:

```
> (my-max 3)
3
> (my-max 3 7 2 9 4)
9
> (my-max -5 -1 -10)
-1
```

The built-in `max` already does this, so name yours `my-max`. Use a named **`let`** to walk the rest arguments.

Exercise 7.5: Tail-Recursive Fibonacci

The naive Fibonacci function shown earlier in this chapter is $O(2^n)$. Rewrite it as a tail-recursive function using two accumulators `a` and `b`:

```
> (fib 0)
0
> (fib 1)
1
> (fib 10)
55
> (fib 50)
12586269025
```

Hint: at each step, the new `a` becomes the old `b`, and the new `b` becomes `a + b`. This mirrors the named **`let`** version from the tail-call section—try writing it from scratch without looking back.

Exercise 7.6: Reversing a List

Write a function `my-reverse` that reverses a list using recursion. Use an accumulator parameter to make it tail-recursive:

```
> (my-reverse '(1 2 3 4 5))
(5 4 3 2 1)
> (my-reverse '())
()
> (my-reverse '(a))
(a)
```

In Python, this would be `lst[::-1]`. Your Scheme version should be O(n) in both time and space.

Exercise 7.7: Appending Two Lists

Write `my-append` that concatenates two lists without using the built-in `append`:

```
> (my-append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
> (my-append '() '(a b))
(a b)
> (my-append '(x) '())
(x)
```

Hint: recurse on the first list. What is the base case? Why does this function copy the first list but share the second?

Exercise 7.8: Deep Reverse

Write a function `deep-reverse` that reverses a list at every level of nesting. Unlike your plain `my-reverse` from earlier, this function must also reverse any sublists it encounters:

```
> (deep-reverse '(1 2 3))
```

```
(3 2 1)
> (deep-reverse '(1 (2 3) (4 (5 6)) 7))
(7 ((6 5) 4) (3 2) 1)
> (deep-reverse '((a b) (c d)))
((d c) (b a))
> (deep-reverse '())
()
```

This requires two recursive cases: when the `car` is itself a pair (deep-reverse it before collecting), and when it is an atom (keep it as-is). Branch on `(pair? (car lst))` the way the `flatten` example from this chapter does, but collect results into an accumulator so each level comes out reversed.

The next chapter covers local bindings in detail—including a closer look at named `let`, which you met here as a looping construct.

Chapter 8
Local Bindings

In Scheme, local variables are introduced with *binding forms*. Unlike Python, where you create a variable simply by assigning to it, Scheme makes the scope of every variable explicit. This chapter covers `let`, `let*`, `letrec`, internal definitions, and the relationship between `let` and `lambda`.

8.1 `let`: Parallel Bindings

`let` introduces one or more local variables:

```
> (let ((x 10)
        (y 20))
    (+ x y))
30
```

The general form is:

```
(let ((var1 expr1)
      (var2 expr2)
      ...)
  body ...)
```

Each `var` is bound to the value of its `expr`, and then the `body` expressions are evaluated in order. The result of the last body expression is the value of the whole `let`.

Bindings Are Parallel

The key rule: all the `expr` values are computed *before* any binding takes effect. This means one binding cannot refer to another:

```
> (let ((x 5)
        (y (* x 2)))   ; ERROR: x is not yet bound here
    y)
error[KP3001]: undefined variable 'x'
```

This fails because `x` is not visible to the expression `(* x 2)`—the bindings are computed in the outer environment. If you need sequential bindings, use `let*`.

Scope Is Limited

Variables introduced by `let` exist only within the body. They do not leak into the surrounding scope:

```
> (define x 100)
> (let ((x 5))
    (display x) (newline)
    x)
5
5
> x
100
```

The outer `x` is temporarily shadowed inside the `let`, but remains unchanged afterward.

Practical Uses

Use `let` to name intermediate results, avoiding repeated computation:

```
> (define (distance x1 y1 x2 y2)
    (let ((dx (- x2 x1))
          (dy (- y2 y1)))
      (sqrt (+ (* dx dx) (* dy dy)))))
> (distance 0 0 3 4)
5
```

8.2 `let*`: Sequential Bindings

`let*` is like `let`, but each binding is visible to subsequent ones. Bindings are evaluated left to right:

```
> (let* ((x 3)
         (y (* x x))
         (z (+ x y)))
    z)
12
```

Here, `y` uses `x` (which is 3), and `z` uses both `x` and `y`.

Coming from Python. `let*` behaves like consecutive variable assignments in Python:

```
x = 3
y = x * x
z = x + y
```

The sequential nature is the same. The difference is that these Scheme bindings are scoped—they vanish after the body, whereas Python variables persist in the enclosing function.

When to Use `let` vs. `let*`

Use `let` when bindings are independent of each other. Use `let*` when later bindings depend on earlier ones. In practice, many programmers default to `let*` since it is never wrong where `let` would work, but using `let` when possible communicates that the bindings are independent.

8.3 `letrec`: Recursive Bindings

`letrec` allows bindings to refer to each other, enabling mutually recursive local functions:

```
> (letrec ((my-even? (lambda (n)
                       (if (= n 0) #t
                           (my-odd? (- n 1)))))
           (my-odd? (lambda (n)
                      (if (= n 0) #f
                          (my-even? (- n 1))))))
    (list (my-even? 10) (my-odd? 10)))
(#t #f)
```

`my-even?` calls `my-odd?` and vice versa. Neither `let` nor `let*` could express this—`let` does not allow forward references, and `let*` only allows references to earlier bindings.

> **Initialization restriction.** The right-hand side expressions in `letrec` must not evaluate the variables being defined during initialization. In practice, this means the expressions should be `lambda` forms (which defer evaluation). Do not write `(letrec ((x (+ y 1)) (y 5)) ...)`—this is undefined behavior.

`letrec*`

`letrec*` is like `letrec` but guarantees left-to-right evaluation order. It is slightly more permissive: later bindings can use the values of earlier bindings (like `let*`), and all bindings can be recursive (like `letrec`):

```
> (letrec* ((x 1)
            (y (+ x 1))
            (f (lambda (n) (+ n x y))))
    (f 10))
13
```

8.4 Named `let`: Loops Revisited

Named `let` was introduced in Chapter 7. Here we examine it more carefully as a binding form.

```
(let name ((var init) ...)
  body ...)
```

This binds `name` to a procedure with parameters `var ...`, calls it with the `init` values, and makes `name` available for recursive calls within the body:

```
> (let loop ((numbers '(1 2 3 4 5))
             (sum 0))
    (if (null? numbers)
        sum
        (loop (cdr numbers)
              (+ sum (car numbers)))))
```

```
15
```

Building Lists with Named `let`

A common pattern is accumulating results:

```
> (define (squares-up-to n)
    (let loop ((i 1) (acc '()))
      (if (> i n)
          (reverse acc)
          (loop (+ i 1) (cons (* i i) acc)))))
> (squares-up-to 5)
(1 4 9 16 25)
```

The accumulator builds the list in reverse (since `cons` adds to the front), and `reverse` fixes the order at the end. This is idiomatic Scheme.

Named `let` vs. `do`

Named `let` and `do` can express the same loops. Named `let` is preferred by most Scheme programmers because the recursive structure is explicit and the control flow is clearer:

```
;; Named let (preferred)
(let loop ((i 0) (sum 0))
  (if (= i 10) sum
      (loop (+ i 1) (+ sum i))))

;; Equivalent do
(do ((i 0 (+ i 1))
     (sum 0 (+ sum i)))
    ((= i 10) sum))
```

Both produce 45. Use whichever reads better for your situation.

8.5 Internal Definitions

Inside a `lambda`, `let`, or `define` body, you can use `define` to create local bindings:

```
> (define (circle-stats radius)
```

```
    (define pi 3.141592653589793)
    (define r-squared (* radius radius))
    (define area (* pi r-squared))
    (define circumference (* 2 pi radius))
    (list area circumference))
> (circle-stats 5)
(78.53981633974483 31.41592653589793)
```

Internal `define`s are equivalent to `letrec*`—they can refer to each other and allow mutual recursion. This is the most common way to define local helper functions:

```
> (define (process-list lst)
    (define (helper x)
      (* x x))
    (map helper lst))
> (process-list '(1 2 3 4))
(1 4 9 16)
```

> **Style preference.** Some programmers prefer internal `define` for local helpers (reads like top-level code), while others prefer `let`/`letrec` (makes scope visually obvious). Both are correct; use what you find clearer.

8.6 `let` Is Syntactic Sugar for `lambda`

Understanding the relationship between `let` and `lambda` deepens your understanding of both. Every `let` can be mechanically translated to an immediately applied `lambda`:

```
;; These two are identical:
(let ((x 5) (y 10))
  (+ x y))

((lambda (x y) (+ x y)) 5 10)
```

The `let` creates an anonymous function with parameters `x` and `y`, then immediately calls it with arguments 5 and 10. This is not just a theoretical curiosity—it explains why `let` bindings are parallel (arguments to a function call are evaluated independently) and why the body has its own scope.

8.7 `let-values`: Destructuring Multiple Values

When a function returns multiple values (using `values`), you can bind them with `let-values`:

```
> (define (min-max lst)
    (let loop ((rest (cdr lst))
               (lo (car lst))
               (hi (car lst)))
      (if (null? rest)
          (values lo hi)
          (loop (cdr rest)
                (min lo (car rest))
                (max hi (car rest))))))
> (let-values (((lo hi) (min-max '(3 1 4 1 5 9 2 6))))
    (display "Min: ") (display lo) (newline)
    (display "Max: ") (display hi) (newline))
Min: 1
Max: 9
```

Two relatives round out the family. `let*-values` is to `let-values` what `let*` is to `let`: later bindings can use earlier ones. And `define-values` binds multiple results as ordinary definitions:

```
> (define-values (lo hi) (min-max '(3 1 4 1 5)))
> lo
1
> hi
5
```

8.8 Choosing the Right Binding Form

Form	When to use
`let`	Independent bindings that do not depend on each other
`let*`	Sequential bindings where later ones depend on earlier ones
`letrec`	Mutually recursive local functions
`letrec*`	Mix of sequential values and recursive functions
Named `let`	Tail-recursive loops with named state
Internal `define`	Local helper functions; equivalent to `letrec*`
`let-values`	Binding multiple return values

8.9 Parameters: Dynamic Binding

Every binding form so far has been *lexical*: a binding is visible only in the source text it encloses. Scheme also offers a controlled form of *dynamic* binding through parameter objects. A parameter is created with `make-parameter` and read by calling it like a function:

```
> (define verbosity (make-parameter 1))
> (verbosity)
1
```

`parameterize` rebinds a parameter *for the duration of its body*—including inside any functions called from that body—and restores the old value afterward:

```
> (define (log-line msg)
    (when (⩾ (verbosity) 2)
      (display msg)
      (newline)))
> (log-line "quiet by default")
> (parameterize ((verbosity 2))
    (log-line "loud inside parameterize"))
loud inside parameterize
> (verbosity)
1
```

Notice what `let` could not do here: `log-line` was defined far away from the `parameterize`, yet it sees the new value. Parameters are how Scheme passes contextual settings—log levels, output destinations, precision—down through many layers of calls without threading an extra argument through every function.

`make-parameter` accepts an optional converter function applied to every value the parameter takes on:

```
> (define timeout (make-parameter "30" string→number))
> (timeout)
30
```

Coming from Python. Parameters resemble Python's `contextvars` (or a disciplined use of thread-local globals): `(parameterize ((p v)) body)` is like setting a context variable inside a `with` block. The value reverts automatically when the body exits—even if it exits with an error.

Several built-in procedures use parameters. In Chapter 10 you will meet `current-output-port` and friends, which are ordinary parameter objects—redirecting output is just a `parameterize`.

Summary

- `let` binds variables in parallel, while `let*` binds them sequentially.
- `letrec` and `letrec*` let bindings refer to one another, as for mutually recursive local functions.
- Named `let` is a concise loop, and internal `define`s behave like `letrec*`.
- `let` is syntactic sugar for an immediately applied `lambda`.
- `let-values`, `let*-values`, and `define-values` bind the results of expressions that return multiple values.
- Parameters provide dynamic binding: `parameterize` changes a value for the extent of a computation, then restores it.

Exercises

Exercise 8.1: Scoping Quiz

Predict the result of each expression below, then check your answers in the REPL. Pay careful attention to which binding form is used:

```
;; Expression A
(let ((x 1))
  (let ((x 2)
        (y x))
    y))

;; Expression B
(let ((x 1))
  (let* ((x 2)
         (y x))
    y))

;; Expression C
(let ((x 10))
  (let ((x (* x 2))
        (y (+ x 3)))
    (list x y)))
```

Why do expressions A and B give different results? What value of x does y see in each case?

Exercise 8.2: Iterative Factorial

Write a function `factorial` using named **`let`** as an iterative loop with an accumulator. Do not use internal **`define`** for a helper—the named **`let`** should be the entire body:

```
> (factorial 0)
1
> (factorial 5)
120
> (factorial 20)
2432902008176640000
```

This should be tail-recursive and handle large inputs thanks to Kaappi's bignum support.

Exercise 8.3: Euclidean GCD

Write a function `euclidean-gcd` that computes the greatest common divisor of two positive integers using the Euclidean algorithm. Use named **`let`** for the loop:

```
> (euclidean-gcd 48 18)
6
> (euclidean-gcd 100 75)
25
> (euclidean-gcd 17 13)
1
```

The algorithm is: if `b` is zero, the GCD is `a`; otherwise, replace `(a, b)` with `(b, a mod b)` and repeat. In Python: `while b: a, b = b, a % b`. The named **`let`** version maps each "iteration" to a tail call.

Kaappi provides a built-in `gcd` procedure—the purpose here is to practice named **`let`**.

Exercise 8.4: Mutual Recursion with `letrec`

Write a function `count-words` that counts the words in a string, where words are separated by one or more whitespace characters. Treat the problem as a tiny state machine with two states, and use **`letrec`** to define one recursive function per state: `between-words` (currently on whitespace) and `in-word` (currently inside a word). Each function examines one character and either stays in its state or hands off to the other:

```
> (count-words "hello world")
2
> (count-words "  a  b  ")
2
> (count-words "scheme")
1
> (count-words "")
0
```

Hint: convert the string to a list of characters with `string→list` and test each with `char-whitespace?`. Count a new word at the moment

`between-words` hands off to `in-word`. Both handoffs are tail calls, so the state machine runs in constant space—two-state problems like this are where mutual recursion genuinely earns its keep.

Exercise 8.5: Rewriting `do` as Named `let`

Rewrite the following **`do`** loop from Chapter 6 as an equivalent named **`let`**. Both should produce the same result:

```
(do ((i 1 (+ i 1))
     (result '() (cons (* i i) result)))
    ((> i 5) (reverse result)))
```

```
> ;; Your named let version should produce:
> (1 4 9 16 25)
```

Which version do you find more readable? Most Scheme programmers prefer named **`let`**—practice translating between the two forms until both feel natural.

In the next chapter, we turn to higher-order functions—procedures that take functions as arguments or return them as results.

Chapter 9
Higher-Order Functions

A *higher-order function* is a function that takes another function as an argument or returns a function as its result. You have already seen examples in earlier chapters—`map`, named `let` with a procedure argument, and closures. This chapter explores the standard higher-order functions in depth and shows how they replace loops in idiomatic Scheme.

> **SRFI-1 imports.** Several procedures in this chapter—`filter`, `remove`, `partition`, `any`, `every`, `fold`, and `fold-right`—come from SRFI-1 (the extended list library), not from `(scheme base)`. In script files, add `(import (srfi 1))` at the top. The REPL makes them available automatically.

9.1 `map`: Transform Every Element

`map` applies a function to every element of a list, returning a new list of the results:

```
> (define (square x) (* x x))
> (map square '(1 2 3 4 5))
(1 4 9 16 25)
> (map string-length '("cat" "elephant" "ox"))
(3 8 2)
> (map not '(#t #f #t #f))
(#f #t #f #t)
```

With a lambda for custom transformations:

```
> (map (lambda (x) (+ x 10)) '(1 2 3))
(11 12 13)
```

```
> (map (lambda (s) (string-append "Hello, " s))
       '("Alice" "Bob" "Carol"))
("Hello, Alice" "Hello, Bob" "Hello, Carol")
```

Multi-List `map`

`map` can take multiple lists, applying the function to corresponding elements:

```
> (map + '(1 2 3) '(10 20 30))
(11 22 33)
> (map * '(2 3 4) '(10 10 10))
(20 30 40)
> (map list '(a b c) '(1 2 3))
((a 1) (b 2) (c 3))
> (map + '(1 2 3) '(10 20))
(11 22)
```

The function receives one argument from each list. If the lists have different lengths, `map` stops at the end of the shortest one, as the last example shows.

> **Coming from Python.** `(map f lst)` is equivalent to Python's `[f(x) for x in lst]` or `list(map(f, lst))`. Multi-list map is like `[f(a, b) for a, b in zip(lst1, lst2)]`.

9.2 `for-each`: Side Effects Only

`for-each` is like `map` but discards the results. Use it when you want side effects (printing, writing to a file) without building a list:

```
> (for-each (lambda (name)
              (display "Hello, ")
              (display name)
              (newline))
            '("Alice" "Bob" "Carol"))
Hello, Alice
Hello, Bob
Hello, Carol
```

`for-each` guarantees left-to-right evaluation order. `map` does not make this guarantee (though most implementations process left to right).

Coming from Python. `for-each` replaces a Python `for` loop where you do not need the return value:

```
for name in ["Alice", "Bob", "Carol"]:
    print(f"Hello, {name}")
```

9.3 `filter`: Select Elements

`filter` keeps only the elements for which a predicate returns true:

```
> (filter even? '(1 2 3 4 5 6 7 8))
(2 4 6 8)
> (filter positive? '(-3 -1 0 2 5 -4 7))
(2 5 7)
> (filter (lambda (s) (> (string-length s) 3))
          '("hi" "hello" "hey" "good morning"))
("hello" "good morning")
```

`remove`: The Complement of `filter`

`remove` (also from SRFI-1) keeps elements that do *not* satisfy the predicate:

```
> (remove even? '(1 2 3 4 5 6))
(1 3 5)
> (remove null? '((1 2) () (3) () (4 5)))
((1 2) (3) (4 5))
```

`partition`: Split into Two Lists

`partition` returns two values: elements that pass and elements that fail:

```
> (let-values (((yes no) (partition even? '(1 2 3 4 5 6))))
    (list yes no))
((2 4 6) (1 3 5))
```

9.4 any and every: Testing Across a List

`any` returns a true value if the predicate holds for at least one element; `every` checks that it holds for all of them:

```
> (any even? '(1 3 5 6))
#t
> (any even? '(1 3 5))
#f
> (every positive? '(1 2 3))
#t
> (every positive? '(1 -2 3))
#f
```

Both short-circuit: `any` stops at the first hit, `every` at the first miss. And like `and` and `or`, `any` returns the predicate's actual result rather than a bare `#t`—useful when the test computes something worth keeping:

```
> (any string→number '("a" "b" "42"))
42
```

> **Coming from Python.** `any` and `every` correspond to Python's `any()` and `all()` over a generator expression: `(any even? lst)` is `any(x % 2 == 0 for x in lst)`.

9.5 fold: Reduce a List to a Single Value

`fold` (also called "reduce" in other languages) combines all elements of a list into one value using a binary function and a starting value:

```
> (fold + 0 '(1 2 3 4 5))
15
> (fold * 1 '(1 2 3 4 5))
120
> (fold max 0 '(3 7 2 9 4))
9
> (fold string-append "" '("a" "b" "c"))
"cba"
```

Why did `string-append` produce `"cba"` instead of `"abc"`? Because `fold` passes each element as the *first* argument and the accumulator as the *second*:

```
;; (fold string-append "" '("a" "b" "c"))
;; Step 1: (string-append "a" "")  = "a"
;; Step 2: (string-append "b" "a") = "ba"
;; Step 3: (string-append "c" "ba") = "cba"
```

The form is `(fold f init lst)`. The function `f` takes two arguments: the current element and the accumulator. Processing is left to right, but the accumulator is the *second* argument:

```
> (fold cons '() '(1 2 3 4 5))
(5 4 3 2 1)
```

Understanding Fold Step by Step

`(fold + 0 '(1 2 3))` evaluates as:

```
;; Start with init = 0
;; Step 1: (+ 1 0) = 1
;; Step 2: (+ 2 1) = 3
;; Step 3: (+ 3 3) = 6
;; Result: 6
```

```
> (fold + 0 '(1 2 3))
6
```

`fold-right`: Right-to-Left Reduction

`fold-right` processes elements from right to left:

```
> (fold-right cons '() '(1 2 3 4 5))
(1 2 3 4 5)
> (fold-right string-append "" '("a" "b" "c"))
"abc"
```

`fold-right` with `cons` copies a list (preserving order), while `fold` with `cons` reverses it.

Expressing Other Operations as Fold

Many list operations are special cases of fold:

```
> ;; sum
> (fold + 0 '(1 2 3 4 5))
15
> ;; length
> (fold (lambda (x acc) (+ acc 1)) 0 '(a b c d))
4
> ;; reverse
> (fold cons '() '(1 2 3))
(3 2 1)
> ;; map (via fold-right)
> (fold-right (lambda (x acc) (cons (* x x) acc))
              '() '(1 2 3 4))
(1 4 9 16)
> ;; filter (via fold-right)
> (fold-right (lambda (x acc)
               (if (even? x) (cons x acc) acc))
             '() '(1 2 3 4 5 6))
(2 4 6)
```

Coming from Python. `fold` is Python's `functools.reduce`. `(fold + 0 '(1 2 3 4 5))` is `reduce(lambda acc, x: acc + x, [1,2,3,4,5], 0)`. But watch the argument order: it is the *reverse* of Python's. `reduce` passes the accumulator first (`acc, x`); `fold` passes the element first (`x acc`). Swapping the two is the classic mistake when porting `reduce` code—if your fold returns a reversed list or a baffling type error, check the argument order first.

9.6 `apply`: Spreading a List as Arguments

`apply`, introduced in Chapter 7, calls a function with the elements of a list as individual arguments—`(apply + '(1 2 3 4 5))` is `15`. One detail that chapter used without comment: you can provide fixed arguments before the list:

```
> (apply + 100 '(1 2 3))
106
> (apply list 'a 'b '(c d e))
```

```
(a b c d e)
```

9.7 Composing Higher-Order Functions

The real power emerges when you combine these operations into pipelines:

```
> ;; Sum of squares of even numbers
> (fold + 0
    (map square
      (filter even? '(1 2 3 4 5 6 7 8 9 10))))
220
```

Read from the inside out: filter the even numbers, square each one, then sum them. In Python this would be:

```
sum(x**2 for x in range(1, 11) if x % 2 == 0)
```

Another example—find the longest string in a list:

```
> (fold (lambda (s longest)
          (if (> (string-length s) (string-length longest))
              s
              longest))
        ""
        '("cat" "elephant" "dog" "hippopotamus"))
"hippopotamus"
```

9.8 Writing Your Own Higher-Order Functions

You are not limited to the standard ones. Here are a few useful patterns:

Applying a Function `n` Times

```
> (define (iterate f n x)
    (if (= n 0) x
        (iterate f (- n 1) (f x))))
> (iterate (lambda (x) (* x 2)) 5 1)
32
> (iterate cdr 3 '(a b c d e))
(d e)
```

Finding the First Match

SRFI-1 already provides this one as `find`—but it is worth seeing what is inside:

```
> (define (my-find pred lst)
    (cond
      ((null? lst) #f)
      ((pred (car lst)) (car lst))
      (else (my-find pred (cdr lst)))))
> (my-find even? '(1 3 5 4 7))
4
> (my-find negative? '(1 2 3))
#f
```

Counting Matches

Also built in as SRFI-1's `count`; here it is as a fold:

```
> (define (my-count pred lst)
    (fold (lambda (x acc)
            (if (pred x) (+ acc 1) acc))
          0 lst))
> (my-count even? '(1 2 3 4 5 6))
3
> (my-count char-alphabetic?
            (string→list "hello, world!"))
10
```

Zipping Two Lists

```
> (define (zip lst1 lst2)
    (map list lst1 lst2))
> (zip '(a b c) '(1 2 3))
((a 1) (b 2) (c 3))
```

Building `filter` from Scratch

The standard higher-order functions are ordinary Scheme code, not magic. Here is `filter`, written with the recursive list pattern from Chapter 7:

```
(define (my-filter pred lst)
  (cond
    ((null? lst) '())
    ((pred (car lst))
     (cons (car lst) (my-filter pred (cdr lst))))
    (else
     (my-filter pred (cdr lst)))))
```

```
> (my-filter even? '(1 2 3 4 5 6))
(2 4 6)
> (my-filter string? '(1 "two" 3 "four"))
("two" "four")
```

Writing `map`, `filter`, or `fold` by hand once is worth the effort: afterward you know exactly what they do, what they cost, and how to build variants when the standard ones do not quite fit.

9.9 When Not to Use Higher-Order Functions

Higher-order functions are elegant, but they are not always the best choice:

- **Complex state:** When the loop body needs to maintain multiple interacting pieces of state, named `let` with explicit accumulators is often clearer than contorting a fold.
- **Early termination:** `map`, `filter`, and `fold` always process the entire list. If you want to stop early, use `find`, `any`, or `every`—they return as soon as they have an answer—or a recursive function.
- **Multiple outputs:** If you need both the filtered elements and the rejected ones, use `partition` rather than calling `filter` twice.

Summary

- `map` transforms every element of a list, while `for-each` applies a procedure for its side effects.
- `filter`, `remove`, and `partition` select elements by a predicate, while `any` and `every` test whether it holds somewhere or everywhere.
- `fold` reduces a list to a single value and can express `map`, `filter`, `length`, and more.

- `apply` spreads a list as arguments, and higher-order functions compose into data-processing pipelines.
- Higher-order functions usually replace explicit recursion, though sometimes plain recursion is clearer.

Exercises

Exercise 9.1: Extracting Data with `map` and `filter`

Given this list of association lists representing people:

```
(define people
  '(((name . "Alice") (age . 30) (city . "Paris"))
    ((name . "Bob") (age . 17) (city . "London"))
    ((name . "Carol") (age . 25) (city . "Paris"))
    ((name . "Dave") (age . 15) (city . "Berlin"))))
```

Write expressions (using `map`, `filter`, and `assq`/`assoc`) that:

1. Extract a list of all names: `("Alice" "Bob" "Carol" "Dave")`
2. Find the names of people aged 18 or older: `("Alice" "Carol")`
3. Find the names of the people who live in Paris: `("Alice" "Carol")`

Exercise 9.2: Fold Implementations

Implement each of these using `fold` (not explicit recursion):

1. `my-length`: returns the number of elements in a list
2. `my-reverse`: reverses a list

```
> (my-length '(a b c d e))
5
> (my-reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Hint: for `my-length`, the accumulator is a counter. For `my-reverse`, the accumulator is a list built with `cons`.

Exercise 9.3: Word Processing Pipeline

Write a function `process-words` that takes a list of strings and:

1. Filters out words shorter than 4 characters
2. Sorts the remaining words alphabetically (use `list-sort` from SRFI-132: `(import (srfi 132))`—sorting is covered properly in Chapter 16; here you only need `(list-sort string<? lst)`)
3. Capitalizes each word (upper-case the first letter, lower-case the rest)

```
> (process-words '("the" "quick" "brown" "fox" "jumps"
                   "over" "the" "lazy" "dog"))
("Brown" "Jumps" "Lazy" "Over" "Quick")
```

Build this as a pipeline: feed the output of `filter` into `list-sort`, then `map` the capitalization function.

Exercise 9.4: Zipping Lists

Implement `zip` that pairs up corresponding elements from two lists. If the lists have different lengths, stop at the shorter one:

```
> (zip '(a b c) '(1 2 3))
((a 1) (b 2) (c 3))
> (zip '(x y) '(10 20 30))
((x 10) (y 20))
```

The chapter showed a one-liner using multi-list `map`, which already stops at the shorter list. Write the recursive version so it behaves the same way—what are the two base cases?

Exercise 9.5: `take` and `drop`

Implement `take` and `drop`, which return the first *n* elements or skip the first *n* elements of a list. Use the SRFI-1 argument order (list first, count second):

```
> (take '(a b c d e) 3)
(a b c)
> (drop '(a b c d e) 3)
(d e)
> (take '(1 2 3) 0)
()
> (drop '(1 2 3) 10)
()
```

Use recursion or named **`let`**. Handle the case where *n* exceeds the list length gracefully. Verify that `(append (take lst n) (drop lst n))` reconstructs the original list.

SRFI-1 provides built-in `take` and `drop`—the purpose here is to practice the recursive pattern.

In the next chapter, we cover input and output—reading from files, writing to the console, and working with ports.

Chapter 10
Input and Output

Programs need to communicate with the outside world: reading user input, writing results to the screen, processing files, and formatting data. Scheme's I/O system is based on *ports*—abstract objects that represent sources and destinations for data. This chapter covers console I/O, file operations, string ports, and the difference between `display` and `write`.

10.1 Output: `display`, `write`, and `newline`

Scheme provides two primary output procedures, each serving a different purpose.

`display`: Human-Readable Output

`display` outputs a value in a form meant for humans. Strings appear without quotes, characters without the `#\` prefix:

```
> (display "Hello, world!")
Hello, world!
> (display 42)
42
> (display #\a)
a
> (display '(1 2 3))
(1 2 3)
```

`display` does not add a newline. Use `newline` to move to the next line:

```
> (begin (display "line one") (newline)
         (display "line two") (newline))
line one
```

```
line two
```

`write`: Machine-Readable Output

`write` outputs a value in a form that can be read back by `read`. Strings include quotes, characters include the `#\` prefix:

```
> (write "hello")
"hello"
> (write #\a)
#\a
> (write '(1 "two" #t))
(1 "two" #t)
```

Use `write` when you want to serialize data that will later be read by a program. Use `display` when you want output meant for a person.

> **Coming from Python.** `display` is like Python's `print( ... , end="")` (without the trailing newline). `write` is like Python's `repr()`—it shows the value in a way you could paste back into code.

Combining Output

A common pattern for formatted output:

```
> (define (show-result label value)
    (display label)
    (display ": ")
    (display value)
    (newline))
> (show-result "Total" 42)
Total: 42
> (show-result "Name" "Alice")
Name: Alice
```

10.2 Input: `read-line` and `read`

`read-line`: Reading Text

`read-line` reads one line of text from standard input (up to the newline, which is consumed but not included):

```
> (define (prompt msg)
    (display msg)
    (flush-output-port)
    (read-line))
> (define name (prompt "Enter your name: "))
Enter your name: Alice
> (string-append "Hello, " name "!")
"Hello, Alice!"
```

The call to `flush-output-port` forces the prompt out of the output buffer so it is visible before `read-line` blocks waiting for input. Output may be buffered until a newline is written or the buffer fills; flush whenever a partial line must appear immediately.

At end of file, `read-line` returns an *eof object*. You can test for it with `eof-object?`:

```
(let loop ((line (read-line)))
  (unless (eof-object? line)
    (display line)
    (newline)
    (loop (read-line))))
```

`read`: Reading Scheme Data

`read` parses a complete Scheme datum from input—a number, string, list, symbol, or any other value the reader understands:

```
> (define x (read))
(1 2 3)
> (car x)
1
> (length x)
3
```

`read` is the inverse of `write`: anything produced by `write` can be read back by `read`. This makes Scheme's S-expression syntax a built-in serialization format.

> **Security.** Be careful with `read` on untrusted input. Unlike Python's `eval`, it cannot execute code—it only builds data structures—but a hostile sender can still exhaust memory with a huge or deeply nested datum. For untrusted data, read a bounded amount with `read-line` and parse the string yourself.

10.3 Ports

A *port* is an abstract source or destination for I/O. Every read operation reads from some port; every write operation writes to some port. When you call `display` or `read-line` without specifying a port, they use the *current ports*:

```
> (current-input-port)
#<input-port stdin>
> (current-output-port)
#<output-port stdout>
> (current-error-port)
#<output-port stderr>
```

You can write to a specific port by passing it as an extra argument:

```
> (display "error!\n" (current-error-port))
error!
```

> **Current ports are parameters.** The current-port procedures are parameter objects (Chapter 8). You can redirect all output within a computation using `parameterize`—no procedure needs to know about the redirection.

Port Predicates

```
> (port? (current-input-port))
#t
> (input-port? (current-input-port))
#t
> (output-port? (current-output-port))
#t
```

10.4 File I/O

Opening and Closing Ports

```
(define in (open-input-file "data.txt"))
;; ... read from in ...
(close-input-port in)

(define out (open-output-file "output.txt"))
;; ... write to out ...
(close-output-port out)
```

Two behaviors to know before you rely on these. Opening an input file that does not exist raises an error—Chapter 14 shows how to catch it with the `file-error?` predicate. And opening an output file that already exists *truncates* it: the old contents are gone the moment the port opens.

`call-with-port`: Safe Port Handling

To ensure ports are always closed (even if an error occurs), use `call-with-port`:

```
(call-with-port (open-input-file "data.txt")
  (lambda (port)
    (let loop ((line (read-line port)))
      (unless (eof-object? line)
        (display line)
        (newline)
        (loop (read-line port))))))
```

The port is automatically closed when the procedure returns.

This open-then-call combination is common enough that R7RS provides shorthands: `call-with-input-file` and `call-with-output-file` take the filename directly:

```
(call-with-input-file "data.txt"
  (lambda (port)
    (read-line port)))
```

`with-input-from-file` and `with-output-to-file`

These redirect the current input/output port for the duration of a thunk (a zero-argument procedure):

```
;; Read all lines from a file
(with-input-from-file "data.txt"
  (lambda ()
    (let loop ((line (read-line)))
      (unless (eof-object? line)
        (display line)
        (newline)
        (loop (read-line))))))
```

```
;; Write to a file
(with-output-to-file "output.txt"
  (lambda ()
    (display "Hello, file!")
    (newline)))
```

> **Coming from Python.** `call-with-port` is similar to Python's `with open( ... ) as f:` context manager—it guarantees the file is closed when you are done.

Reading an Entire File

A utility that reads all lines into a list:

```
(define (read-all-lines filename)
  (call-with-port (open-input-file filename)
    (lambda (port)
      (let loop ((line (read-line port))
                 (lines '()))
        (if (eof-object? line)
            (reverse lines)
            (loop (read-line port)
                  (cons line lines)))))))
```

Writing to a File

```
(define (write-lines filename lines)
  (call-with-port (open-output-file filename)
    (lambda (port)
      (for-each (lambda (line)
                  (display line port)
                  (newline port))
                lines))))
```

`file-exists?` and `delete-file`

Two housekeeping procedures round out file handling. Here we use the `write-lines` helper from above to create a file, then remove it:

```
> (write-lines "data.txt" '("some" "content"))
> (file-exists? "data.txt")
#t
> (delete-file "data.txt")
> (file-exists? "data.txt")
#f
```

Check `file-exists?` before opening a file for output when silently overwriting it would be destructive.

10.5 String Ports

String ports let you use I/O procedures on strings instead of files. They are useful for building strings incrementally or parsing string content.

Output String Ports

```
> (define port (open-output-string))
> (display "Hello, " port)
> (display "world!" port)
> (get-output-string port)
"Hello, world!"
```

This is more efficient than repeated `string-append` for building large strings.

Input String Ports

```
> (define port (open-input-string "(+ 1 2) (+ 3 4)"))
> (read port)
(+ 1 2)
> (read port)
(+ 3 4)
> (read port)
#<eof>
```

Input string ports are useful for parsing structured data from strings.

Capturing Output as a String

A handy pattern using an output string port:

```
> (define (value→string val)
    (let ((port (open-output-string)))
      (write val port)
      (get-output-string port)))
> (value→string '(1 2 3))
"(1 2 3)"
> (value→string "hello")
"\"hello\""
```

10.6 The EOF Object

When a read operation reaches the end of input, it returns an *eof object*—a special value distinct from all other Scheme values:

```
> (define port (open-input-string ""))
> (read-line port)
#<eof>
> (eof-object? (read-line port))
#t
```

Always test for EOF using `eof-object?`. The standard loop pattern is:

```
(let loop ((datum (read port)))
  (unless (eof-object? datum)
    ;; process datum
```

```
    (loop (read port))))
```

10.7 `read-char` and `write-char`

For character-level I/O:

```
> (define port (open-input-string "hello"))
> (read-char port)
#\h
> (read-char port)
#\e
> (peek-char port)
#\l
> (read-char port)
#\l
```

`peek-char` returns the next character without consuming it—useful for lookahead in parsers.

`write-char` writes a single character to a port:

```
> (write-char #\A)
A
> (write-char #\newline)
```

Between whole lines and single characters, `read-string` reads up to *k* characters as a string, and `write-string` writes a string (or a substring) to a port:

```
> (define port (open-input-string "hello world"))
> (read-string 5 port)
"hello"
> (read-string 100 port)
" world"
```

10.8 Binary Ports

Every port seen so far is *textual*: it reads and writes characters. R7RS also defines *binary* ports, which carry raw bytes. Open them with `open-binary-input-file` and `open-binary-output-file`, and move data with `read-u8`, `write-u8`, `read-bytevector`, and `write-bytevector`:

```
;; Read the first four bytes of a file (e.g. to check
;; a format signature)
(define (file-magic filename)
  (call-with-port (open-binary-input-file filename)
    (lambda (port)
      (read-bytevector 4 port))))
```

Real PNG files start with the signature bytes 137, 80, 78, 71. We can fabricate a minimal test file with `write-bytevector` and check it:

```
> (call-with-port (open-binary-output-file "photo.png")
    (lambda (port)
      (write-bytevector (bytevector 137 80 78 71 13 10 26 10)
                        port)))
> (file-magic "photo.png")
#u8(137 80 78 71)
```

Binary reads return bytevectors (Chapter 4) or, at end of file, the same EOF object as textual reads. A port is either textual or binary, never both—calling `read-line` on a binary port is an error. Use binary ports for images, compressed data, and network protocols; use textual ports for anything meant to be read by humans.

10.9 Scripts: Arguments, Environment Variables, and Exit Codes

Every example so far has named its own files. Real command-line tools receive their inputs from the outside: arguments, environment variables, and—in the other direction—an exit status for the shell. R7RS covers all three in the `(scheme process-context)` library.

`command-line` returns the program's arguments as a list of strings, with the script name first. Save this as `args.scm`:

```
(import (scheme base) (scheme write)
        (scheme process-context))

(display (command-line))
(newline)
```

```
kaappi args.scm one two
```

```
(args.scm one two)
```

Your arguments are `(cdr (command-line))`—the equivalent of Python's `sys.argv[1:]`.

`exit` ends the program immediately with a status code for the shell: `(exit)` or `(exit 0)` reports success, and any nonzero value, like `(exit 1)`, reports failure. There is also `emergency-exit`, which terminates without running the cleanup actions registered by `dynamic-wind` (Chapter 15); prefer `exit` unless you know you need that.

`get-environment-variable` looks up one environment variable, returning its string value, or `#f` when it is not set:

```
> (get-environment-variable "HOME")
"/Users/bmuthuka"
> (get-environment-variable "NO-SUCH-VARIABLE")
#f
```

Its companion `get-environment-variables` returns the entire environment as an association list of name–value string pairs, ready for `assoc` (Chapter 5).

> **Coming from Python.** `(command-line)` is `sys.argv`, and `(exit 1)` is `sys.exit(1)`. `(get-environment-variable "X")` behaves like `os.environ.get("X")`, down to returning a falsy value when the variable is missing.

10.10 Putting It Together: A File Processing Example

Here is a program that reads a text file, numbers each line, and writes the result to a new file:

```
(define (number-lines input-file output-file)
  (call-with-port (open-input-file input-file)
    (lambda (in)
      (call-with-port (open-output-file output-file)
        (lambda (out)
          (let loop ((line (read-line in))
                     (n 1))
            (unless (eof-object? line)
```

```
              (display n out)
              (display ": " out)
              (display line out)
              (newline out)
              (loop (read-line in) (+ n 1)))))))))
```

Usage, with `write-lines` and `read-all-lines` from earlier in the chapter:

```
> (write-lines "input.txt" '("alpha" "beta" "gamma"))
> (number-lines "input.txt" "output.txt")
> (for-each (lambda (l) (display l) (newline))
            (read-all-lines "output.txt"))
1: alpha
2: beta
3: gamma
```

To turn this into a standalone tool, add a driver that reads the two filenames from `command-line` and complains—on the error port, with a failing exit status—when they are missing. Save the definition of `number-lines` followed by this driver as `number-lines.scm`:

```
(define args (cdr (command-line)))

(unless (= (length args) 2)
  (display "usage: kaappi number-lines.scm INPUT OUTPUT"
           (current-error-port))
  (newline (current-error-port))
  (exit 1))

(number-lines (car args) (cadr args))
```

```
kaappi number-lines.scm input.txt output.txt
```

The shell sees exit status 0 on success and 1 on a usage error, which makes the script a well-behaved citizen in pipelines and Makefiles.

Summary

- `display` produces human-readable output, while `write` produces machine-readable output that `read` can parse back.

- `read-line` reads a line of text and `read` reads a single Scheme datum.
- All I/O flows through ports; `call-with-port` and the `with-*-to-file` forms open and close them safely.
- String ports perform in-memory I/O, including capturing output as a string.
- Binary ports carry raw bytes as bytevectors; textual ports carry characters. A port is one or the other.
- `command-line`, `get-environment-variable`, and `exit` from `(scheme process-context)` connect a script to its arguments, environment, and exit status.
- Input procedures return the EOF object when input is exhausted.

Exercises

Exercise 10.1: A Simple `cat`

Write a function `cat` that takes a filename and prints its contents to the console, line by line. Use `call-with-port` to ensure the file is properly closed:

```
(define (cat filename)
  ;; your code here
  )
```

Test it by creating a small text file and running `(cat "test.txt")` in the REPL. This is the Scheme equivalent of the Unix `cat` command (or Python's `print(open(f).read())`). As a bonus, turn it into a standalone script: take the filename from `command-line`, and exit with status 1 (after printing a usage message) when no filename is given or the file does not exist (`file-exists?`).

Exercise 10.2: Word Count

Write a function `wc` that takes a filename and returns three values (using `values`): the number of lines, words, and characters in the file. Use `read-line` in a loop to process the file line by line:

```
> (let-values (((lines words chars) (wc "example.txt")))
    (display lines) (display " lines, ")
    (display words) (display " words, ")
    (display chars) (display " characters")
    (newline))
5 lines, 23 words, 142 characters
```

Hint: for counting words in a line, use Kaappi's built-in `string-split` (a Kaappi extension to SRFI-13—always available, no import needed), or count transitions between whitespace and non-whitespace characters.

Exercise 10.3: File to List

Write a function `file→lines` that reads all lines from a file and returns them as a list of strings. The chapter showed a `read-all-lines` helper—try writing it from memory:

```
> (file→lines "example.txt")
("first line" "second line" "third line")
```

Then write the inverse: `lines→file` that takes a filename and a list of strings and writes each string as a line. Verify round-tripping: `(file→lines f)` after `(lines→file f data)` should return `data`.

Exercise 10.4: String Port Serialization

Use string ports to implement `serialize` and `deserialize` functions that convert Scheme values to strings and back:

```
> (define s (serialize '(1 "hello" #t (nested list))))
> s
"(1 \"hello\" #t (nested list))"
> (deserialize s)
```

```
(1 "hello" #t (nested list))
```

`serialize` should use `open-output-string` and `write`. `deserialize` should use `open-input-string` and `read`. Test with various types: numbers, strings, booleans, nested lists, and symbols. Why is `write` (not `display`) essential for round-tripping?

This completes Part II of the book. In Part III, we move to advanced topics, starting with Scheme's powerful macro system (Chapter 11).

λ

Part III

Going Further

With the core language in hand, this part covers the tools you reach for as programs grow: macros that extend Scheme's own syntax, the library system for organizing code into modules, user-defined record types, structured error handling, and continuations. Read it once you are comfortable writing functions and working with lists.

Chapter 11
Macros

Macros are what make Lisp languages uniquely powerful. They let you extend the language itself—defining new syntactic forms that look and behave exactly like built-in ones. If you find yourself writing the same boilerplate pattern repeatedly, a macro can eliminate it.

Scheme's macro system is *hygienic*: variables introduced by a macro never accidentally capture variables from the surrounding code. This eliminates an entire class of bugs that plague macro systems in C and Common Lisp.

This chapter covers `define-syntax` and `syntax-rules`—the pattern-based macro system specified by R7RS.

11.1 What Is a Macro?

A macro is a compile-time transformation rule. When Kaappi encounters a macro call, it does not evaluate the arguments—instead, it rewrites the entire expression according to the macro's rules, and then evaluates the result.

One boundary is worth stating up front: macros transform *expressions*, the tree of parenthesized forms the reader has already parsed. They cannot change how characters are read in the first place. Some Lisps let you extend the reader itself with new lexical syntax; Kaappi (like standard R7RS) does not—the token syntax of Chapter 3 is fixed.

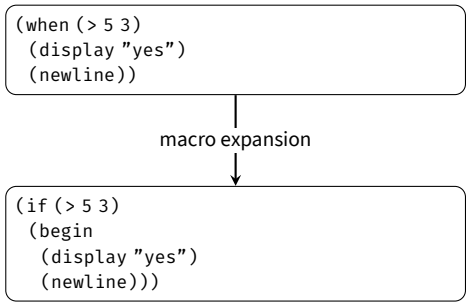
Consider `when`:

```
> (when (> 5 3)
    (display "yes")
    (newline))
yes
```

`when` is not a built-in—it is a macro that expands to an `if`:

```
(if (> 5 3)
    (begin
      (display "yes")
      (newline)))
```

The macro takes the *source code* and transforms it into different source code before evaluation:



Coming from Python. Python has no equivalent feature. Decorators modify functions but cannot create new syntax. Macros operate at the level of code itself—they generate code from code.

11.2 Your First Macro

Define a macro with `define-syntax` and `syntax-rules`:

```
> (define-syntax my-when
    (syntax-rules ()
      ((my-when test body ...)
       (if test (begin body ...)))))
> (my-when (> 5 3)
    (display "hello")
    (newline))
hello
```

Let's break this down:

- `define-syntax` names the macro (`my-when`).
- `syntax-rules ()` introduces the transformation rules. The `()` is a list of literal keywords (empty here).

- `(my-when test body ...)` is the *pattern*—it matches the macro call.
- `(if test (begin body ...))` is the *template*—what the macro expands to.
- `...` (ellipsis) means "zero or more of the preceding element."

11.3 Seeing the Expansion

When a macro misbehaves, look at what it expands to. The Kaappi REPL's `,expand` command shows the expansion of any expression without evaluating it:

```
> ,expand (when #t (display "yes"))
(if #t (begin (display "yes")))
> ,expand (my-when (> 5 3) (display "hello") (newline))
(if (> 5 3) (begin (display "hello") (newline)))
```

Make `,expand` a habit while working through this chapter: after each macro you write, expand a sample call and check that the generated code is what you intended. It turns macro debugging from guesswork into reading. The same view exists outside the REPL: `kaappi expand file.scm` prints a whole program after macro expansion, which is handy when a macro misbehaves only in the middle of a real file.

11.4 Pattern Matching

`syntax-rules` patterns work by structural matching:

- A bare identifier (like `test` or `body`) matches any single expression and binds it as a *pattern variable*.
- An identifier followed by `...` matches zero or more expressions.
- Literal parentheses match literal parentheses in the input.
- Identifiers listed in the literals list are matched by name, not as pattern variables.
- The underscore `_` is a wildcard: it matches any single form without binding it. Because the macro name in a call always matches trivially, real-world macros conventionally write the name position as `_`: `((_ test body ...) ...)`. This book spells the name out for readability, but expect `_` in code you read elsewhere.

Multiple Clauses

A macro can have multiple patterns. The first matching pattern wins:

```
> (define-syntax my-and
    (syntax-rules ()
      ((my-and) #t)
      ((my-and x) x)
      ((my-and x rest ...)
       (if x (my-and rest ...) #f))))
> (my-and)
#t
> (my-and 1 2 3)
3
> (my-and 1 #f 3)
#f
```

This is a recursive macro: the third clause expands to another call to `my-and` with fewer arguments, until it reaches a base case.

If no clause matches a call, expansion fails with a compile-time syntax error at the call site. The `dotimes` macro later in this chapter shows how to catch a misuse deliberately with `syntax-error`.

Literal Keywords

The first argument to `syntax-rules` is a list of identifiers to match literally rather than as pattern variables:

```
> (define-syntax my-if
    (syntax-rules (then else)
      ((my-if test then consequent else alternate)
       (cond (test consequent)
             (else alternate)))))
> (my-if (> 3 2) then "yes" else "no")
"yes"
```

Here, `then` and `else` are literal keywords—they must appear in the call exactly as written, not as variable names.

11.5 The Ellipsis (...)

The ellipsis is the key to writing macros that accept variable numbers of subforms.

In a pattern, `expr ...` matches zero or more expressions. In a template, `expr ...` replicates the template for each matched element:

```
> (define-syntax my-list
    (syntax-rules ()
      ((my-list) '())
      ((my-list elem rest ...)
       (cons elem (my-list rest ...)))))
> (my-list 1 2 3)
(1 2 3)
```

A more useful example—a macro that returns the value of the first expression while evaluating the rest for side effects:

```
> (define-syntax begin0
    (syntax-rules ()
      ((begin0 first rest ...)
       (let ((result first))
         rest ...
         result))))
> (begin0 (+ 1 2)
    (display "side effect")
    (newline))
side effect
3
```

Nested Ellipsis

Ellipsis can be nested for macros that process lists of lists:

```
> (define-syntax my-let
    (syntax-rules ()
      ((my-let ((var init) ...) body ...)
       ((lambda (var ...) body ...) init ...))))
> (my-let ((x 5) (y 10))
    (+ x y))
15
```

This `my-let` macro transforms into a lambda application—exactly how `let` is defined in the language.

11.6 Hygiene

Scheme macros are hygienic: temporary variables introduced by a macro cannot clash with variables in the calling code. Consider:

```
> (define-syntax swap!
    (syntax-rules ()
      ((swap! a b)
       (let ((tmp a))
         (set! a b)
         (set! b tmp)))))
> (let ((tmp 100) (x 1) (y 2))
    (swap! x y)
    (list tmp x y))
(100 2 1)
```

The `tmp` inside the macro and the `tmp` in the calling code are different bindings—the macro's `tmp` is automatically renamed to avoid capture. In an unhygienic system (like C's `#define`), this would be a bug.

There is no magic here, and `,expand` lets you watch the mechanism work. During expansion, every identifier that comes from the *template* (rather than from the macro call) is renamed to a fresh, globally unique name:

```
> ,expand (swap! x y)
(__hyg_1_let ((__hyg_2_tmp x)) (set! x y) (set! y __hyg_2_tmp))
```

Read the expansion alongside the template. The template's `tmp` became `__hyg_2_tmp`, so it can never collide with the `tmp` at the call site. The `x` and `y` came from the macro *call*, not the template, so they keep their names and refer to the caller's bindings. Even `let` was renamed—the expander renames every template identifier uniformly, and the compiler recognizes renamed references to standard forms—which means the macro still works if the caller has shadowed `let` with a local variable. (The numbers in the fresh names are a session-wide counter, so yours will differ.)

Hygiene, then, is just consistent renaming: template identifiers are renamed away from the caller's namespace, call-site identifiers pass through untouched.

Why hygiene matters. In C, the macro `#define SWAP(a,b) {int tmp=a; a=b; b=tmp;}` breaks if you write `SWAP(tmp, x)` because the

user's `tmp` and the macro's `tmp` collide. Scheme's system makes this impossible.

11.7 Practical Macros

A `unless` Macro

Both `when` and `unless` are built-in, but they are defined as macros internally. Here is how `unless` could be implemented:

```
(define-syntax unless
  (syntax-rules ()
    ((unless test body ...)
     (if (not test) (begin body ...)))))
```

A `while` Loop

Scheme does not have a built-in `while`, but you can add one:

```
> (define-syntax while
    (syntax-rules ()
      ((while test body ...)
       (let loop ()
         (when test
           body ...
           (loop))))))
> (let ((i 0))
    (while (< i 5)
      (display i) (display " ")
      (set! i (+ i 1))))
0 1 2 3 4
```

A `dotimes` Loop

```
> (define-syntax dotimes
    (syntax-rules ()
      ((dotimes (var count) body ...)
       (let loop ((var 0))
         (when (< var count)
           body ...
```

```
          (loop (+ var 1)))))))
> (dotimes (i 5)
    (display (* i i))
    (display " "))
0 1 4 9 16
```

A call that does not match the pattern—say `(dotimes 5 ...)`, forgetting the parentheses around `(i 5)`—fails with a syntax error at expansion time. You can make such misuse explicit by adding a catch-all clause that expands to `syntax-error`, an R7RS form that signals an error during expansion:

```
(define-syntax dotimes
  (syntax-rules ()
    ((dotimes (var count) body ...)
     (let loop ((var 0))
       (when (< var count) body ... (loop (+ var 1)))))
    ((dotimes other ...)
     (syntax-error "dotimes: expected (variable count)"))))
```

The catch-all clause matches anything the first clause rejected, and the `syntax-error` in its template stops compilation. The message string also documents, right in the macro definition, what shape of call was expected.

An Assertion Macro

```
> (define-syntax assert
    (syntax-rules ()
      ((assert expr)
       (unless expr
         (error "Assertion failed" 'expr)))))
> (assert (= (+ 1 1) 2))
> (assert (= (+ 1 1) 3))
error[KP3000]: Assertion failed (= (+ 1 1) 3)
```

The failing assertion raises an error object whose message is `"Assertion failed"` and whose irritant is the *source code* of the assertion—`'expr` in the template quotes it, which is why the report shows the exact expression that failed. A handler can read both parts back; Chapter 14 shows how.

A Timing Macro

This macro reports how long an expression takes to evaluate. `current-second`, from the standard `(scheme time)` library (Chapter 16), returns the current time in seconds:

```
(define-syntax time-it
  (syntax-rules ()
    ((time-it expr)
     (let ((start (current-second)))
       (let ((result expr))
         (let ((elapsed (- (current-second) start)))
           (display "Elapsed: ")
           (display elapsed)
           (display "s")
           (newline)
           result))))))
```

11.8 Local Macros with `let-syntax`

You can define macros with limited scope using `let-syntax`:

```
> (let-syntax ((double (syntax-rules ()
                        ((double x) (+ x x)))))
    (double 21))
42
```

The macro `double` exists only within the body. For mutually recursive macros, use `letrec-syntax`.

> **Macro arguments may be evaluated more than once.** A `syntax-rules` macro substitutes its arguments as *code*, not as computed values. If a pattern variable appears several times in the template, its argument expression runs once for each occurrence. A `square` macro whose body is `(* x x)` expands `(square (read))` into `(* (read) (read))`, reading its input twice. When an argument may have side effects or is expensive, bind it once with `let` inside the macro before reusing it.

11.9 When to Use Macros

Macros are powerful but should not be overused. Use a macro when:

- You need to control *when* or *whether* arguments are evaluated (like `when`, `unless`, `and`).
- You need to introduce new bindings (like `let`, `dotimes`).
- You want to generate code at compile time to avoid runtime overhead.
- You are building a domain-specific language (DSL) with custom syntax.

Do *not* use a macro when a function would suffice. Functions are simpler, composable (you can pass them to `map`), and easier to debug. If your "macro" does not need to delay evaluation or introduce bindings, make it a function.

Summary

- Macros transform code at expansion time, before evaluation, letting you add new syntactic forms.
- `syntax-rules` matches input against patterns and rewrites it using output templates.
- The ellipsis (`...`) matches and expands sequences of forms and can be nested.
- Scheme macros are hygienic, so introduced identifiers never accidentally capture the user's variables.
- The REPL's `,expand` command shows what a call expands to—the first tool to reach for when a macro misbehaves.
- `let-syntax` defines local macros; reach for macros only when a function cannot do the job.

Exercises

> **Exercise 11.1: Swap!**
>
> Write a `swap!` macro that swaps the values of two variables using `set!`. Verify that hygiene prevents the macro's temporary variable from clashing with user code.

```
> (let ((a 1) (b 2))
    (swap! a b)
    (list a b))
(2 1)
> (let ((tmp 99) (x 10) (y 20))
    (swap! x y)
    (list tmp x y))
(99 20 10)
```

Exercise 11.2: Assert-Equal

Write an `assert-equal` macro that takes two expressions, evaluates both, and raises an error showing the source code of each expression and their values if they are not `equal?`. (Hint: use `'expr` in the template to capture source code.)

```
> (assert-equal (+ 1 2) 3)
;; no output --- assertion passed
> (assert-equal (* 2 3) 7)
error[KP3000]: assertion failed: (* 2 3) evaluated to 6,
    expected 7
```

Exercise 11.3: Do-Times with Named Counter

Write a **`do`**`-n-times` macro similar to the `dotimes` macro from this chapter, but the user supplies the counter variable name, the count, *and* an optional step size (defaulting to 1). You will need two `syntax-rules` clauses.

```
> (do-n-times (i 5)
    (display i) (display " "))
0 1 2 3 4
> (do-n-times (i 10 3)
    (display i) (display " "))
0 3 6 9
```

Exercise 11.4: With-Logging

Write a `with-logging` macro that prints an entry message before evaluating the body and an exit message after, then returns the body's result. The macro should accept a label string as its first argument.

```
> (with-logging "compute"
    (+ 1 2))
[enter: compute]
[exit: compute]
3
```

As a bonus, make it print the elapsed time in the exit message using `current-second`.

In the next chapter (Chapter 12), we cover the library system—how to organize code into modules with clean interfaces.

Chapter 12
The Library System

As programs grow, you need to organize code into separate files with clean interfaces. Scheme's R7RS library system provides `define-library` for creating modules and `import` for using them. This chapter covers importing standard libraries, writing your own, controlling exports, and structuring multi-file projects.

12.1 Importing Libraries

Every Scheme program begins by importing the libraries it needs:

```
(import (scheme base)
        (scheme write))

(display "Hello, world!")
(newline)
```

Strictly speaking, Kaappi does not force you to write these imports: every built-in procedure is globally available, in script files as well as in the REPL. Write them anyway. The import list documents which parts of the language a program uses, and other R7RS implementations *require* it—code without imports is not portable. Imports also genuinely matter within Kaappi for the *portable* SRFI libraries described below, which are unavailable until imported. You can import any number of libraries in a single `import` form. Notice that we import `(scheme write)` above: in R7RS, `display` and `write` are formally defined there, not in `(scheme base)`.

Available Standard Libraries

R7RS defines 16 standard libraries. The 15 you will actually use are listed below; the sixteenth, `(scheme r5rs)`, exists only for running legacy R5RS code:

Library	Contents
`(scheme base)`	Core procedures: arithmetic, lists, strings, vectors, I/O, control
`(scheme write)`	`write`, `display`, `write-shared`
`(scheme read)`	`read`
`(scheme char)`	Unicode character procedures
`(scheme file)`	File port operations
`(scheme eval)`	`eval`, `environment`
`(scheme repl)`	`interaction-environment`
`(scheme load)`	`load`
`(scheme inexact)`	Transcendental functions (`sin`, `cos`, `sqrt`, etc.)
`(scheme complex)`	Complex number operations
`(scheme lazy)`	`delay`, `force`, promises
`(scheme time)`	`current-second`, jiffies
`(scheme case-lambda)`	`case-lambda`
`(scheme cxr)`	`cadr`, `caddr`, etc. (24 compositions)
`(scheme process-context)`	`exit`, `command-line`, environment variables

Beyond these, Kaappi provides 72 SRFIs importable as `(srfi N)`:

```
> (import (srfi 1))   ; Extended list library
> (filter odd? '(1 2 3 4 5))
(1 3 5)
> (import (srfi 69))  ; Hash tables
> (import (srfi 13))  ; String library
```

The REPL also accepts the `,import` command, which takes the same forms: `,import (srfi 1)`.

Built-in vs. Portable SRFIs

The 72 SRFIs come in two kinds, and the difference shows up in practice:

- **Eight built-in SRFIs** (1, 9, 13, 18, 39, 69, 133, 170) are implemented inside the runtime and always available—`filter` from SRFI-1 works even if you forget the import.
- **64 portable SRFIs** are ordinary `.sld` libraries loaded on demand. These *must* be imported: `(list-sort < lst)` fails with an undefined-variable error until you write `(import (srfi 132))`.

When in doubt, write the import—it is always correct, and it keeps the program portable. Appendix C catalogs all 72 and marks which are built in.

12.2 Import Modifiers

Sometimes you want to import only specific names, avoid conflicts, or add a prefix. Import modifiers give you fine-grained control.

> **Coming from Python.** Scheme's `import` brings names in *unqualified* by default—closer to Python's `from module import *` than to `import module`. There is no `module.name` access; every imported binding becomes a plain top-level name. The modifiers below recover the control Python gives you: `only` is like `from m import a, b`; `rename` is like `from m import a as b`; and `prefix` gives you qualified-looking names such as `ht:ref`.

`only`: Import Specific Names

```
(import (only (scheme base) map filter define let if))
```

Only the listed names become available. Everything else from the library is excluded.

`except`: Exclude Specific Names

```
(import (except (scheme base) error))
```

Everything from `(scheme base)` is imported except `error`.

`prefix`: Add a Namespace Prefix

```
> (import (prefix (scheme char) char:))
> (char:char-alphabetic? #\A)
#t
> (char:char-upcase #\a)
#\A
```

All imported names get the prefix `char:`. This avoids name collisions between libraries.

`rename`: Rename Specific Imports

```
(import (rename (scheme base) (define def) (lambda fn)))
(def square (fn (x) (* x x)))
```

Individual names can be renamed on import. This is useful when two libraries export the same name.

Combining Modifiers

Modifiers can be nested:

```
(import (prefix (only (srfi 1) fold filter partition) list:))
;; Now available as: list:fold, list:filter, list:partition
```

> **Name conflicts are resolved silently.** If two imported libraries export the same name, Kaappi does not raise an error—the *last* import wins, and the earlier binding is silently shadowed. That makes accidental clashes easy to miss. When two libraries you need both export the same name, make your intent explicit with `only`, `except`, `prefix`, or `rename`.

12.3 Writing Your Own Library

A library is defined in a `.sld` file (Scheme Library Definition). The file name corresponds to the library name: the library `(mylib math)` lives in `mylib/math.sld`.

Basic Structure

```
(define-library (mylib math)
  (export square cube factorial)
  (import (scheme base))
  (begin
    (define (square x) (* x x))
```

```
    (define (cube x) (* x x x))

    (define (factorial n)
      (let loop ((i n) (acc 1))
        (if (= i 0) acc
            (loop (- i 1) (* i acc)))))))
```

The key declarations:

- `export` — lists the names that users of this library can access.
- `import` — lists libraries that this library depends on.
- `begin` — contains the Scheme code that defines the exported bindings.

Using Your Library

Create a program that imports it:

```
(import (mylib math))

(display (square 5))
(newline)
(display (factorial 10))
(newline)
```

Run from the directory containing the `mylib/` folder:

```
kaappi program.scm
```

```
25 3628800
```

12.4 Library Search Paths

Kaappi searches for `.sld` files in this order:

1. The current working directory (`./`)
2. The `./lib/` subdirectory
3. Directories specified with `-lib-path` (repeatable, up to 16)

4. The directory containing the script being run
5. The user library directory (`~/.kaappi/lib/`)

The library name `(mylib utils)` maps to the file path `mylib/utils.sld`. Each component of the library name becomes a directory, except the last which becomes the filename.

To add custom search paths:

```
kaappi --lib-path ./vendor/libs --lib-path ./mylibs program.scm
```

12.5 Export Specifications

Simple Exports

```
(export square cube factorial)
```

Each listed name becomes available to importers.

Renamed Exports

You can export a binding under a different name:

```
(define-library (mylib strings)
  (export (rename internal-join join)
          (rename internal-split split))
  (import (scheme base))
  (begin
    (define (internal-join lst sep) ...)
    (define (internal-split s sep) ...)))
```

Users import `join` and `split`, but internally the functions are named `internal-join` and `internal-split`.

Re-Exporting Imported Bindings

A library can export names it imported from another library. This supports a *facade*: one library presenting a single interface assembled from several internal ones:

```
(define-library (mylib all)
  (export parse tokenize join split)
  (import (scheme base)
```

```
          (mylib parser)      ; provides parse, tokenize
          (mylib strings)))   ; provides join, split
```

Users import `(mylib all)` and get the combined interface, while you remain free to reorganize the internal libraries behind it.

12.6 The `include` Declaration

For libraries with large implementations, you can keep the code in separate files:

```
(define-library (mylib parser)
  (export parse tokenize)
  (import (scheme base) (scheme char))
  (include "parser-impl.scm"))
```

The file `parser-impl.scm` is read as if its contents were inside a `begin` block. This keeps `.sld` files short and focused on the interface.

12.7 `cond-expand`: Conditional Code

`cond-expand` lets you include different code based on which features are available:

```
(define-library (mylib platform)
  (export get-os)
  (import (scheme base))
  (cond-expand
    (kaappi
     (begin
       (define (get-os) "kaappi")))
    (else
     (begin
       (define (get-os) "unknown")))))
```

Kaappi defines the feature identifiers `kaappi`, `r7rs`, `ieee-float`, `posix`, `exact-closed`, and `exact-complex`, plus one identifier per optional subsystem—`kaappi-fibers`, `kaappi-reactor`, `kaappi-threads`, and `kaappi-diagnostics`—so portable code can test for green threads or OS threads (Chapter 18) the same way it tests for an operating system. The built-in procedure `(features)` returns the list the running interpreter reports:

```
> (features)
(r7rs kaappi ieee-float posix exact-closed exact-complex
    kaappi-fibers kaappi-reactor kaappi-diagnostics
    kaappi-threads)
```

12.8 Project Structure

A well-organized Kaappi project looks like this:

```
my-project/
  main.scm              # Entry point
  mylib/
    core.sld            # (mylib core)
    utils.sld           # (mylib utils)
    http/
      client.sld        # (mylib http client)
      server.sld        # (mylib http server)
```

```
;; main.scm
(import (mylib core)
        (mylib utils)
        (mylib http client))

(start-app)
```

Avoiding Circular Dependencies

Libraries cannot have circular imports. If library A imports library B, then B cannot import A. If you encounter this, extract the shared definitions into a third library C that both A and B import.

12.9 Packages with Thottam

For sharing libraries with others, Kaappi uses the `thottam` package manager. A package is a library (or set of libraries) published to the package registry:

```
thottam install kaappi-json
```

After installation, you can import the library directly:

```
(import (kaappi json))
```

Packages are installed to `~/.kaappi/lib/`, which is always on the search path. We will use several ecosystem packages in Part IV. For reference material, see Appendix B (the full `thottam` command set) and Appendix E (the catalog of available packages).

Summary

- `import` brings library bindings into scope, and the modifiers `only`, `except`, `prefix`, and `rename` control exactly what is imported.
- Kaappi's built-ins (including eight built-in SRFIs) are globally available, but the 64 portable SRFIs and your own libraries genuinely require `import`—and portable R7RS code always writes its imports.
- `define-library` declares a library with explicit `export` and `import` sets.
- Libraries are found through the search path, and `include` and `cond-expand` support file inclusion and conditional code.
- The `thottam` package manager publishes and installs third-party packages.

Exercises

Exercise 12.1: Create a Utility Library

Create a file `mylib/utils.sld` that defines a library `(mylib utils)` exporting three utility procedures: `clamp` (restricts a number to a min/max range), `repeat-string` (returns a string repeated *n* times), and `displayln` (displays a value followed by a newline). Write a short program that imports and uses all three.

```
;; mylib/utils.sld
(define-library (mylib utils)
  (export clamp repeat-string displayln)
  (import (scheme base) (scheme write))
  (begin
    ;; your definitions here
    ))
```

Test in the REPL:

```
> (clamp 15 0 10)
10
> (repeat-string "ha" 3)
"hahaha"
```

Exercise 12.2: Import Modifiers

Import SRFI-1 (the extended list library) three different ways in a single program using import modifiers:

1. Use **`only`** to import just `fold` and `unfold`.
2. Use **`prefix`** to import the entire library with a `list:` prefix.
3. Use **`rename`** to import `partition` as `split-by`.

Write a short program that demonstrates all three imports working. Which approach would you choose for a large project, and why?

Exercise 12.3: Conditional Code with cond-expand

Write a library `(mylib greeting)` that uses **`cond`**`-expand` to export a `greet` procedure. When the `kaappi` feature is present, `greet` should mention it; on other Scheme implementations it should return a generic greeting. As a bonus, add an `(`**`and`**` kaappi posix)` branch that also includes the login name from `(get-environment-variable "USER")`.

```
> (import (mylib greeting))
> (greet "Alice")
"Hello, Alice! (running on Kaappi, user: baiju)"
```

In the next chapter (Chapter 13), we define custom data types with records.

λ

Chapter 13
Records and Data Abstraction

As programs grow, you need structured data with named fields—not just anonymous lists where you have to remember that "the third element is the email address." Scheme's `define-record-type` creates custom data types with constructors, predicates, and accessors. This chapter covers record definition, usage patterns, and how to build clean abstractions.

13.1 Why Records?

Consider representing a point as a list:

```
> (define p (list 3 4))
> (car p)   ; x? y? who knows?
3
```

This works but has problems: there is no way to distinguish a point from any other two-element list, no way to tell which element is `x` and which is `y` without reading the code that created it, and no error if you accidentally pass a string where a point is expected.

Records solve all three problems.

13.2 Defining a Record Type

```
> (define-record-type <point>
    (make-point x y)
    point?
    (x point-x)
    (y point-y))
> (define p (make-point 3 4))
```

```
> (point? p)
#t
> (point-x p)
3
> (point-y p)
4
> (point? '(3 4))
#f
```

The `define-record-type` form takes:

1. **Type name** — `<point>` (the angle brackets are a convention, not syntax).
2. **Constructor** — `(make-point x y)` specifies the constructor name and which fields it initializes.
3. **Predicate** — `point?` tests whether a value is an instance of this record type.
4. **Field specifications** — each field is `(name accessor)` or `(name accessor mutator)`.

13.3 Accessors and Mutators

Each field gets an accessor. You can optionally add a mutator for mutable fields:

```
> (define-record-type <counter>
    (make-counter value)
    counter?
    (value counter-value set-counter-value!))
> (define c (make-counter 0))
> (counter-value c)
0
> (set-counter-value! c 10)
> (counter-value c)
10
```

Fields without a mutator are immutable—once set by the constructor, they cannot be changed:

```
> (point-x p)
3
> ;; No set-point-x! exists --- the field is immutable
```

Prefer immutable records. Make fields immutable by default (no mutator). Immutable data is easier to reason about, safe to share between threads, and less prone to bugs. Add mutators only when you genuinely need in-place updates.

13.4 Records vs. Lists

Aspect	Lists	Records
Type checking	Not possible (`pair?` matches all lists)	Unique predicate per type
Field access	By position (`car`, `cadr`)	By name (`point-x`)
Self-documenting	No	Yes
Extensibility	Easy (just cons more)	Fixed at definition
Pattern matching	Via `car`/`cdr`	Via predicate + accessors

Use lists for homogeneous collections (a list of numbers). Use records for heterogeneous structured data (a person with a name, age, and email).

Coming from Python. `define-record-type` plays the role of a Python `class` or `@dataclass`: a named type with named fields, a constructor, and a predicate (like `isinstance`). One difference to watch for: unlike a `@dataclass`, two records with identical field values are *not* automatically equal—`(equal? a b)` is `#f` unless `a` and `b` are the very same object. Write your own comparison procedure when you need value equality. This matters most in lookups. `member` and `assoc` accept an equality predicate as an optional last argument—`(member p points point=?)` finds points by value. SRFI-69 hash tables (Chapter 16) offer no such hook: keys always compare with `equal?`, so a record key matches only the identical object. To look up records by content, key the table with plain data instead, such as a list of field values.

13.5 A Larger Example: A Task Manager

```
(define-record-type <task>
  (make-task title priority done?)
  task?
  (title task-title)
```

```
  (priority task-priority)
  (done? task-done? set-task-done!))
```

```
> (define tasks
    (list (make-task "Write chapter" 'high #f)
          (make-task "Review PR" 'medium #f)
          (make-task "Buy milk" 'low #t)))
> (filter (lambda (t) (not (task-done? t))) tasks)
(#<<task> "Write chapter" high #f> #<<task> "Review PR" medium
    #f>)
> (task-title (car tasks))
"Write chapter"
> (set-task-done! (car tasks) #t)
> (task-done? (car tasks))
#t
```

Records print as `#«task> ... >` with their field values in definition order, which makes them easy to inspect at the REPL.

13.6 Constructors with Fewer Fields

The constructor does not have to initialize all fields. You can list only the fields that should be set at creation time:

```
(define-record-type <entry>
  (make-entry key value)
  entry?
  (key entry-key)
  (value entry-value)
  (timestamp entry-timestamp set-entry-timestamp!))
```

Here, `make-entry` only takes `key` and `value`. The `timestamp` field is uninitialized until explicitly set. Accessing an uninitialized field returns an unspecified value—so be careful with this pattern.

A safer approach is to always initialize every field:

```
(define (create-entry key value)
  (let ((e (make-entry key value)))
    (set-entry-timestamp! e (current-second))
    e))
```

(`current-second`, from the standard `(scheme time)` library, returns the current time in seconds; Chapter 16 covers it.)

13.7 Building Abstractions with Records

Records shine when combined with procedures that form a clean interface. Here is a simple stack:

```
(define-record-type <stack>
  (make-stack-internal items)
  stack?
  (items stack-items set-stack-items!))

(define (make-stack) (make-stack-internal '()))

(define (stack-push! s item)
  (set-stack-items! s (cons item (stack-items s))))

(define (stack-pop! s)
  (let ((items (stack-items s)))
    (when (null? items)
      (error "stack underflow"))
    (set-stack-items! s (cdr items))
    (car items)))

(define (stack-empty? s)
  (null? (stack-items s)))
```

```
> (define s (make-stack))
> (stack-push! s 10)
> (stack-push! s 20)
> (stack-push! s 30)
> (stack-pop! s)
30
> (stack-pop! s)
20
> (stack-empty? s)
#f
```

Users of this stack never access `stack-items` directly—they use the provided interface. The internal representation can change without breaking client code.

13.8 Immutable Records and Functional Updates

For immutable records, the “update” pattern creates a new record with one field changed:

```
(define-record-type <person>
  (make-person name age email)
  person?
  (name person-name)
  (age person-age)
  (email person-email))

(define (person-with-age p new-age)
  (make-person (person-name p)
               new-age
               (person-email p)))
```

```
> (define alice (make-person "Alice" 30 "alice@example.com"))
> (define older-alice (person-with-age alice 31))
> (person-age alice)
30
> (person-age older-alice)
31
```

The original record is unchanged. This functional update style is safe for concurrent code and easy to reason about.

> **Coming from Python.** This pattern is like `dataclasses.replace(obj, age=31)` or `NamedTuple._replace()`. Scheme has no built-in syntax for it, so you write the helper yourself.

13.9 Printing Records

As the task manager showed, records print as the type name followed by the field values in definition order:

```
> (make-point 3 4)
#<<point> 3 4>
```

This is handy at the REPL, but the field names are not shown, the format is not standard across Scheme implementations, and `read` cannot parse it back. For user-facing output, define a display procedure:

```
(define (display-point p)
  (display "(")
  (display (point-x p))
  (display ", ")
  (display (point-y p))
  (display ")"))
```

```
> (display-point (make-point 3 4))
(3, 4)
```

13.10 Records and Dispatch

Scheme does not have built-in pattern matching on records (unlike Haskell or Rust). The idiomatic approach is *predicate dispatch*: test with the predicate and destructure with accessors. Consider two shape types:

```
(define-record-type <circle>
  (make-circle radius)
  circle?
  (radius circle-radius))

(define-record-type <rectangle>
  (make-rectangle width height)
  rectangle?
  (width rectangle-width)
  (height rectangle-height))
```

A dispatch function tests each predicate and extracts the relevant fields:

```
(define (describe shape)
  (cond
    ((circle? shape)
     (string-append "circle with radius "
                    (number→string (circle-radius shape))))
    ((rectangle? shape)
     (string-append "rectangle "
                    (number→string (rectangle-width shape))
```

```
                    "x"
                    (number→string (rectangle-height shape))))
    (else "unknown shape")))
```

```
> (describe (make-circle 5))
"circle with radius 5"
> (describe (make-rectangle 3 4))
"rectangle 3x4"
```

This works well for small type hierarchies. For larger ones, consider a dispatch table keyed on predicates.

13.11 When to Use Records

- **Structured data with named fields** — configuration objects, database rows, parsed messages.
- **Type-safe dispatch** — when you need to distinguish between different kinds of values (`circle?` vs. `rectangle?`).
- **Encapsulation** — hide internal representation behind a constructor + accessor interface.
- **Self-documenting code** — `(person-name p)` is clearer than `(car p)`.
- **Custom error types** — Chapter 14 uses records to build structured, catchable exception objects.

Do not use records when a simple list or pair suffices (e.g., a coordinate pair used only locally) or when the structure needs to vary at runtime (use association lists instead).

Summary

- `define-record-type` creates a distinct type with a constructor, a predicate, accessors, and optional mutators.
- Records provide named fields and type safety, unlike positional lists.
- A constructor need not accept every field, which supports records with defaults.

- Functional updates produce new records instead of mutating, enabling immutable designs.

- Without built-in pattern matching, records are dispatched on with predicates and destructured with accessors.

Exercises

Exercise 13.1: Book Records

Define a `<book>` record type with fields for `title`, `author`, `year`, and `isbn`. All fields should be immutable. Write a `display-book` procedure that formats a book as a readable string.

```
> (define b (make-book "SICP" "Abelson & Sussman" 1996
                       "978-0262510875"))
> (book? b)
#t
> (display-book b)
SICP by Abelson & Sussman (1996) [978-0262510875]
```

Exercise 13.2: Mutable Bank Account

Define a `<bank-account>` record with an immutable `owner` field and a mutable `balance` field. Write `deposit!` and `withdraw!` procedures. The `withdraw!` procedure should raise an error if the withdrawal would cause a negative balance.

```
> (define acct (make-bank-account "Alice" 100))
> (deposit! acct 50)
> (account-balance acct)
150
> (withdraw! acct 30)
> (account-balance acct)
120
> (withdraw! acct 200)
error[KP3000]: insufficient funds: 200 exceeds balance 120
```

Exercise 13.3: Functional Update Helper

Using the `<book>` record from Exercise 1, write a general-purpose `book-with-year` procedure that returns a new book identical to the original but with a different year. Then write `book-with-title` the same way. Can you see why a general "copy-with" pattern would be useful? (Python solves this with `dataclasses.replace()`.)

```
> (define b1 (make-book "SICP" "Abelson" 1985 "000"))
> (define b2 (book-with-year b1 1996))
> (book-year b1)
1985
> (book-year b2)
1996
> (book-title b2)
"SICP"
```

Exercise 13.4: Priority Queue ADT

Build a priority queue abstraction using records. Define a `<pqueue>` record that internally stores items as a sorted list. Implement `make-pqueue`, `pqueue-insert` (returns a new priority queue with the item added), `pqueue-peek` (returns the highest-priority item), and `pqueue-pop` (returns a new priority queue with the highest-priority item removed). Lower numbers mean higher priority.

```
> (define pq (make-pqueue))
> (define pq2 (pqueue-insert pq 3 "low"))
> (define pq3 (pqueue-insert pq2 1 "high"))
> (define pq4 (pqueue-insert pq3 2 "med"))
> (pqueue-peek pq4)
"high"
> (define pq5 (pqueue-pop pq4))
> (pqueue-peek pq5)
"med"
```

In the next chapter (Chapter 14), we cover error handling—how to signal, catch, and recover from errors.

λ

Chapter 14
Error Handling

Programs encounter errors: files that do not exist, invalid user input, network failures, division by zero. Scheme provides a structured exception system with `guard` for catching errors, `raise` for signaling them, and `error` for creating error objects. This chapter covers all three, plus strategies for writing robust code. When you hit an error you do not recognize, Appendix D lists the most common messages along with their causes and fixes.

14.1 The `error` Procedure

The simplest way to signal an error is the `error` procedure:

```
> (error "something went wrong")
error[KP3000]: something went wrong
```

`error` creates an *error object* and raises it as an exception. The first argument is a message string; additional arguments are *irritants*—values that provide context about what went wrong, as in `(error "index out of range" 5)`.

At the top level, the REPL prints the message and irritants and abandons the evaluation. They are not just display text: they travel inside the error object, and any handler that catches it can read them back—as `guard` does in the next section. The `[KP3000]` tag is a *diagnostic code*: every error Kaappi reports carries one, and Section 14.2 shows how to put them to work.

Using `error` in Your Functions

```
> (define (safe-divide a b)
    (if (zero? b)
        (error "division by zero" a b)
```

```
        (/ a b)))
> (safe-divide 10 2)
5
> (safe-divide 10 0)
error[KP3000]: division by zero 10 0
```

Note how the irritants `10` and `0` appear after the message—this is why you pass the offending values to `error` instead of formatting them into the string.

What an Uncaught Error Looks Like

When no handler catches an error, Kaappi reports it and abandons the evaluation. Errors detected by the runtime itself—type errors, arity mismatches, undefined variables—are reported with their full detail, and running a script also gets you the file, line, and column of the offending expression, the source line itself, and the call site it was reached from:

```
$ kaappi process.scm
process.scm:2:3: error[KP3002]: type error in 'car': expected pair, got 5
      (car n))
  called from process.scm:3
```

Learn to read these reports before learning to catch errors—most of the time, the right response to an unexpected error is to fix the bug it points at, not to handle it.

> **Coming from Python.** `(error "message" value)` is like Python's `raise ValueError(f"message: {value}")`. The key difference is that Scheme's error objects are structured—the message and irritants are separate fields, not formatted into a string.

14.2 Catching Errors with `guard`

`guard` is Scheme's primary exception-handling form. It catches raised exceptions and matches them against clauses, similar to `try/except` in Python:

```
> (guard (e (#t (string-append "caught: "
                               (error-object-message e))))
    (error "file not found" "data.txt"))
"caught: file not found"
```

The `#t` clause here matches *any* raised object—fine for a first demonstration, but deliberately blunt. Real handlers should name the errors they expect, as the rest of this chapter shows (and as the warning on catch-all handlers below insists).

The general form:

```
(guard (variable
        (test1 expr1 ...)
        (test2 expr2 ...)
        ...
        (else expr ...))
  body ...)
```

The `body` is evaluated. If an exception is raised, it is bound to `variable`, and the clauses are tested in order (like `cond`). The first matching clause's expressions are evaluated and returned. If no clause matches, the exception is re-raised.

Multiple Clauses

```
> (guard (e
          ((string? e)
           (string-append "string error: " e))
          ((number? e)
           (* e 10))
          (else "unknown error"))
    (raise "oops"))
"string error: oops"
```

Inspecting Error Objects

Error objects created by `error` can be examined with three procedures:

```
> (guard (e (#t (list
                 (error-object? e)
                 (error-object-message e)
                 (error-object-irritants e))))
    (error "bad value" 42 'expected-string))
(#t "bad value" (42 expected-string))
```

- `error-object?` — tests if the value is an error object
- `error-object-message` — extracts the message string

- `error-object-irritants` — extracts the list of irritant values

`file-error?` and `read-error?`

R7RS singles out two error classes with dedicated predicates. `file-error?` recognizes errors raised by file operations—most commonly, opening a file that does not exist. `read-error?` recognizes errors raised by `read` on malformed input:

```
> (guard (e ((file-error? e) 'no-such-file))
    (open-input-file "missing.txt"))
no-such-file
> (guard (e ((read-error? e) 'bad-syntax))
    (read (open-input-string ")")))
bad-syntax
```

Prefer these predicates over matching on message text. Error messages differ between Scheme implementations—and may change between Kaappi versions—but the predicates are part of the standard.

Conditional Error Handling

You can match on the error message or type to handle different errors differently:

```
(define (safe-lookup key table)
  (guard (e
          ((and (error-object? e)
                (string=? (error-object-message e)
                          "key not found"))
           #f)
          (else (raise e)))  ; re-raise unknown errors
    (lookup key table)))
```

The `(else (raise e))` clause ensures that unexpected errors are not silently swallowed.

Matching on the message string works here because `lookup` is our own code and we control the text. It is brittle for anything else: an error message is not part of a procedure's contract and may change without notice. For errors from other code, dispatch on a predicate instead—`file-error?`, `read-error?`, a custom error type as shown later in this chapter—or, for errors raised by the interpreter itself, on the stable code shown next.

> **Always re-raise unknown errors.** Do not write `(guard (e (#t #f)) ...)` to blindly catch everything. This hides bugs. Only catch the specific errors you know how to handle, and re-raise the rest.

Stable Error Codes

Every diagnostic Kaappi reports carries a stable code—the `KP3002` in `error[KP3002]: type error in 'car'`—and the built-in `(kaappi diagnostics)` library reads it back programmatically. `error-object-code` returns the code as a symbol, or `#f` for a value that has none:

```
> (import (kaappi diagnostics))
> (guard (e (#t (error-object-code e)))
    (car '()))
KP3002
> (guard (e (#t (error-object-code e)))
    (error "bad input" 42))
#f
```

Note the second result: codes identify *interpreter* diagnostics. Error objects you create with `error` carry no code—the `KP3000` the printer shows for them means only "user-raised error"—so inside your own code you still dispatch on messages or custom error types. Codes shine when you must catch one specific runtime failure without depending on its message text:

```
> (guard (e ((eq? (error-object-code e) 'KP3004)
             'caught-division))
    (/ 1 0))
caught-division
```

`error-object-code` is total—it accepts any value, error object or not, and never raises—so it is safe as the first test in a `guard` clause. And when a code in a report means nothing to you, ask the interpreter what it stands for:

```
$ kaappi explain KP3004
KP3004  division-by-zero
runtime · error

division by zero

An exact division, 'modulo', 'remainder', or 'quotient' had a zero
```

```
divisor. Guard the divisor, or use inexact arithmetic where the result
is a floating-point infinity or NaN instead of an error.
```

Appendix D walks through the most common codes; `kaappi explain --all` lists every one.

14.3 Raising Exceptions with `raise`

`raise` signals an exception. Any value can be raised—it does not have to be an error object:

```
> (guard (e (#t e))
    (raise "a string error"))
"a string error"
> (guard (e (#t e))
    (raise 42))
42
> (guard (e (#t e))
    (raise '(custom error data)))
(custom error data)
```

However, the convention is to use `error` (which creates a proper error object) for most situations. Direct `raise` with arbitrary values is useful for custom condition types.

`raise` vs. `error`

Form	Use when
`(error msg irritants ...)`	Signaling an error with a message—the common case
`(raise value)`	Signaling a non-error condition, or raising a custom condition object

14.4 `with-exception-handler`: Low-Level Handler

`with-exception-handler` installs an exception handler for the duration of a thunk:

```
> (call/cc
    (lambda (exit)
      (with-exception-handler
```

```
        (lambda (e) (exit (list 'caught e)))
        (lambda () (raise "boom")))))
(caught "boom")
```

This example leans on `call/cc`, which is not covered until Chapter 15. For now, read `(call/cc (lambda (exit) ...))` as "capture an early-return procedure named `exit`"—calling `exit` abandons the rest of the computation and returns its argument.

`with-exception-handler` is lower-level than `guard`. The handler receives the exception and must decide what to do: it can invoke an escape continuation, re-raise, or (for continuable exceptions) return a value.

In most code, `guard` is preferred because it is clearer and handles re-raising automatically. Use `with-exception-handler` only when you need the full power (e.g., logging all exceptions without catching them).

14.5 Continuable Exceptions

`raise-continuable` raises an exception that the handler can "fix" by returning a value:

```
> (with-exception-handler
    (lambda (e) (* e 10))    ; handler returns a replacement
    (lambda () (+ 1 (raise-continuable 5))))
51
```

The handler receives 5, returns 50, and execution continues as if `raise-continuable` had returned 50. This is an advanced feature—use it for situations where a handler can supply a default or corrected value.

14.6 Common Patterns

Default Values on Error

Here we intentionally catch all errors—any failure (missing file, parse error, or otherwise) means we fall back to the default:

```
(define (read-config-or-default filename)
  (guard (e (#t '()))  ; return empty list on any error
    (call-with-port (open-input-file filename)
      (lambda (port) (read port)))))
```

Guaranteed Cleanup

For guaranteed resource cleanup (similar to Python's `try/finally`), Scheme provides `dynamic-wind`. Its third thunk runs no matter how the body exits—normally or via an exception:

```
> (guard (e (#t (display "caught\n")))
    (dynamic-wind
      (lambda () (display "enter\n"))
      (lambda () (error "boom"))
      (lambda () (display "cleanup\n"))))
enter
cleanup
caught
```

Chapter 15 explains `dynamic-wind` in full, including why it takes three thunks. For the common case of file handling, the simplest approach remains `call-with-port`, which closes the port when the body returns or raises an exception.

Wrapping External Calls

```
(define (safe-json-parse str)
  (guard (e
          ((error-object? e)
           (values #f (error-object-message e)))
          (else
           (values #f "unknown parse error")))
    (values (json-read (open-input-string str)) #f)))
```

This returns two values: the parsed result (or `#f` on error) and an error message (or `#f` on success).

Asserting Preconditions

```
(import (srfi 132))   ; provides list-sort

(define (vector-median v)
  (when (zero? (vector-length v))
    (error "cannot compute median of empty vector" v))
  (let ((sorted (list-sort < (vector→list v))))
    (list-ref sorted (quotient (length sorted) 2))))
```

Fail early with a clear message when preconditions are not met.

14.7 Custom Error Types

For larger programs, you may want to distinguish between different kinds of errors. One approach is to use records as error objects and raise them with `raise`. We use `define-record-type` (covered in Chapter 13) to define a custom error type with a type name, a constructor, a predicate, and field specifications:

```
(define-record-type <http-error>
  (make-http-error status message)
  http-error?
  (status http-error-status)
  (message http-error-message))

(define (fetch url)
  (let ((response (http-get url)))
    (unless (= (response-status response) 200)
      (raise (make-http-error
               (response-status response)
               (response-body response))))
    (response-body response)))
```

Catch them with `guard` using the record predicate:

```
> (guard (e
          ((http-error? e)
           (string-append "HTTP "
             (number→string (http-error-status e))
             ": " (http-error-message e)))
          ((error-object? e)
           (error-object-message e)))
    (fetch "https://example.com/missing"))
"HTTP 404: Not Found"
```

This pattern gives you type-safe error dispatch—the `guard` clauses test each error type in order, and unrecognized errors fall through to the next clause or propagate up.

14.8 Error Handling in Libraries

When writing a library, follow these conventions so that callers can handle your errors cleanly:

- **Use `error` with a consistent prefix.** Start every error message with the library or procedure name: `(error "json-read: unexpected token" ch)`. This makes it immediately clear where the error originated.
- **Include the relevant values as irritants.** The caller needs context to diagnose the problem. Include the input that caused the error, the expected type, and the actual value.
- **Document which errors your procedures raise.** If `json-read` can raise a parse error, say so in the documentation. Callers should not have to read your source code to know what to catch.
- **Do not catch errors you cannot handle.** If a low-level I/O function fails inside your library, let that error propagate. Only catch errors that your library can meaningfully recover from.

```
(define (parse-config filename)
  (guard (e ((file-error? e)
             (error "parse-config: file not found" filename)))
    (call-with-port (open-input-file filename)
      (lambda (port) (json-read port)))))
```

This wraps the low-level file error in a domain-specific message while preserving the offending filename as an irritant. Note the use of `file-error?` rather than matching the message text—the predicate keeps working even if the underlying message changes.

14.9 Hunting Bugs with the Debugger

`guard` and `raise` manage the errors you expect. For the bug you do not understand yet—a procedure returning the wrong value, with no error in sight—the REPL has an interactive debugger. `,break` sets a breakpoint on any procedure. The next time it is called, evaluation pauses and drops you into a `debug>` prompt where you can look around:

```
(define (average lst)
```

```
  (/ (apply + lst) (length lst)))

(define (report lst)
  (display (average lst))
  (newline))
```

```
> ,break average
Breakpoint set on average
> (report '(3 4 8))
Break at average (<repl>:1)
debug> locals
  lst = (3 4 8)
debug> backtrace
> [2] average
  [1] report
  [0] <lambda>
debug> continue
5
```

`locals` shows the paused procedure's bindings—exactly what `average` received, with no `display` calls sprinkled through your code. `backtrace` shows how execution got here, innermost frame first (the > marks the current frame): the expression typed at the REPL called `report`, which called `average`. `continue` resumes the program as if nothing had happened.

From the same prompt, `step` moves into the next expression, `next` steps over it, and `finish` runs to the end of the current procedure. `watch var` pauses when a variable changes—useful when some piece of state mutates and you cannot see where from. When you are done, `,delete all` clears every breakpoint. The full command set for both the REPL and the `debug>` prompt is tabulated in Appendix B.

The debugger complements, rather than replaces, the habits from this chapter: read the error report first (it usually names the culprit), reproduce the problem with the smallest input you can, and only then reach for a breakpoint to watch the wrong value being computed.

14.10 Error Handling Strategy

- **Fail fast.** Use `error` to signal problems as soon as they are detected. Do not pass invalid data deeper into the program hoping something will work.

- **Catch at boundaries.** Handle errors at the boundary between subsystems—at the top of a request handler, at the entry point of a CLI tool, at the edge of a library call. Do not pepper every function with `guard`.
- **Be specific.** Catch only the errors you can handle. Let unexpected errors propagate—they usually indicate bugs that should not be silently swallowed.
- **Provide context.** Use irritants in `error` to include the values that caused the problem. "index out of range" is less useful than "index out of range: 10, list length: 5."
- **Log, then re-raise.** When you need to observe errors without handling them, install a handler that logs and re-raises:

```
(define (with-error-logging thunk)
  (guard (e (#t (display "ERROR: " (current-error-port))
                (display (if (error-object? e)
                             (error-object-message e)
                             e)
                         (current-error-port))
                (newline (current-error-port))
                (raise e)))
    (thunk)))
```

Summary

- `error` signals a condition with a message and irritants, while `raise` signals any object.
- `guard` catches exceptions with `cond`-style clauses, and `with-exception-handler` is the underlying primitive.
- Dispatch on predicates, not message text: `file-error?` and `read-error?` identify the standard error classes.
- Records make good custom error types, letting handlers distinguish one kind of error from another.
- Common patterns include supplying default values on error, guaranteeing cleanup, and asserting preconditions.

- For bugs rather than errors, `,break` pauses a running program at a procedure so you can inspect `locals` and the `backtrace` before continuing.

Exercises

Exercise 14.1: Safe Division

Write a `safe-divide` procedure that takes a numerator, a denominator, and an optional default value. If the denominator is zero, it returns the default (or `0` if none is provided) instead of raising an error. Use **`guard`** to catch the error.

```
> (safe-divide 10 3)
10/3
> (safe-divide 10 0)
0
> (safe-divide 10 0 -1)
-1
```

Exercise 14.2: Retry on Error

Write a `retry` procedure that takes a thunk and a maximum number of attempts. It calls the thunk; if it raises an error, it tries again up to *n* times. If all attempts fail, it re-raises the last error. Print which attempt is being made.

```
> (let ((count 0))
    (retry (lambda ()
             (set! count (+ count 1))
             (if (< count 3)
                 (error "not yet" count)
                 count))
           5))
Attempt 1 failed: not yet
Attempt 2 failed: not yet
3
```

Exercise 14.3: Validate with Descriptive Errors

Write a `validate-email` procedure that checks a string for basic email validity. Raise descriptive errors with irritants for each failure case: the string must contain exactly one `@` character, must have at least one character before the `@`, and must have a `.` after the `@`. Each error message should include the input string as an irritant.

```
> (validate-email "alice@example.com")
#t
> (validate-email "no-at-sign")
error[KP3000]: missing @ in email address "no-at-sign"
> (validate-email "@example.com")
error[KP3000]: empty local part in email address
    "@example.com"
```

Exercise 14.4: With-Default

Write a `with-default` procedure that takes a default value and a thunk. It calls the thunk; if any error is raised, it returns the default value instead. This is similar to Python's pattern of `try: ... except: return default`.

```
> (with-default 0 (lambda () (+ 1 2)))
3
> (with-default 0 (lambda () (error "boom")))
0
> (with-default '() (lambda () (vector-ref #(1 2) 5)))
()
```

In the next chapter (Chapter 15), we explore continuations—Scheme's most powerful and unusual control flow mechanism.

Chapter 15
Continuations

Continuations are Scheme's most distinctive feature. A continuation represents "the rest of the computation"—everything that would happen after a particular point in the program. Scheme lets you capture this as a first-class value and invoke it later, enabling non-local exits, coroutines, backtracking, and custom control structures.

This chapter starts with simple escape continuations (early returns), then covers full continuations, and finishes with `dynamic-wind` for cleanup guarantees.

15.1 What Is a Continuation?

At every point during execution, there is a "continuation"—the rest of the work to be done. Consider:

```
(* 3 (+ 1 2))
```

When Scheme evaluates `(+ 1 2)`, the continuation is "take the result and multiply it by 3":

The dashed box is the "hole" where the value of `(+ 1 2)` will be plugged in. Normally, continuations are implicit—you never see them. Scheme's `call/cc` makes them explicit.

15.2 Escape Continuations with `call/ec`

The most common use of continuations is an *early return*—exiting a computation before it would normally finish. `call/ec` (call with escape continuation) provides this:

```
> (call/ec (lambda (return)
             (return 42)
             (display "never reached")))
42
```

When `return` is invoked, execution immediately jumps out of the `call/ec` form with the given value. Code after the call is skipped.

Early Return from a Loop

A common pattern: searching a list and returning the first match:

```
> (define (find-negative lst)
    (call/ec (lambda (return)
      (for-each (lambda (x)
                  (when (negative? x) (return x)))
                lst)
      #f)))
> (find-negative '(3 1 -4 1 5))
-4
> (find-negative '(1 2 3))
#f
```

Without `call/ec`, you would need recursion or a flag variable. The escape continuation gives you the equivalent of Python's `return` inside a loop.

Breaking Out of Nested Computation

```
> (define (product lst)
    (call/ec (lambda (bail)
      (let loop ((rest lst) (acc 1))
        (cond
          ((null? rest) acc)
          ((zero? (car rest)) (bail 0))
          (else (loop (cdr rest) (* acc (car rest)))))))))
> (product '(1 2 3 4 5))
120
```

```
> (product '(1 2 0 4 5))
0
```

If we see a zero, we bail immediately—no point multiplying the remaining elements.

> **Coming from Python.** `call/ec` is like wrapping code in a function and using `return`. Scheme does not have a `return` statement, so `call/ec` fills that role. The important difference: it works across function boundaries (the escape continuation can be invoked from inside a lambda passed to `for-each`).

15.3 Full Continuations with `call/cc`

`call/cc` (short for `call-with-current-continuation`) captures the *full* continuation—not just an escape, but a snapshot of the entire execution state that can be invoked multiple times, even after the original computation has finished.

```
> (call/cc (lambda (k) (k 42)))
42
> (+ 1 (call/cc (lambda (k) (+ 2 (k 3)))))
4
```

In the second example, the continuation of `call/cc` is "add 1 to the result." Calling `(k 3)` jumps to that point with 3, so the answer is 4. The `(+ 2 ...)` is abandoned.

Saving and Reusing a Continuation

The real power of `call/cc` is that you can save the continuation and invoke it later:

```
> (define saved #f)
> (define (capture)
    (call/cc (lambda (k)
      (set! saved k)
      "first time")))
> (capture)
"first time"
> (saved "second time")
"second time"
```

```
> (saved "third time")
"third time"
```

Each call to `saved` re-enters the computation at the point where `call/cc` was called, as if it had returned the given value.

A Counter with Continuations

```
> (define resume #f)
> (define (start)
    (let ((result (call/cc (lambda (k)
                    (set! resume k)
                    0))))
      (display result)
      (display " ")
      (when (< result 3)
        (resume (+ result 1)))))
> (start)
0 1 2 3
```

Each call to `resume` re-enters the `let`, with `result` bound to the new value. This is how coroutines and generators can be implemented.

15.4 `call/ec` vs. `call/cc`

Feature	`call/ec`	`call/cc`
Invoke after proc returns?	No	Yes
Invoke multiple times?	No	Yes
Stack snapshot?	No	Yes
Performance	Fast	Slow (copies stack)
Use case	Early return	Coroutines

> **Prefer `call/ec`.** Use `call/ec` unless you genuinely need to reinvoke the continuation after the procedure returns. Full `call/cc` copies the entire call stack, which is O(stack depth). For early exits and error handling, `call/ec` is dramatically cheaper.

15.5 dynamic-wind: Guaranteed Cleanup

`dynamic-wind` ensures that setup and cleanup code run whenever control enters or leaves a region—even when continuations jump in or out:

```
(dynamic-wind
  before-thunk    ; called when entering
  body-thunk      ; the main computation
  after-thunk)    ; called when leaving
```

```
> (let ((log '()))
    (dynamic-wind
      (lambda () (set! log (cons 'enter log)))
      (lambda () (set! log (cons 'body log)))
      (lambda () (set! log (cons 'leave log))))
    (reverse log))
(enter body leave)
```

Resource Cleanup

The most common use is ensuring resources are released:

```
(define (with-open-file filename proc)
  (let ((port (open-input-file filename)))
    (dynamic-wind
      (lambda () #f)
      (lambda () (proc port))
      (lambda () (close-input-port port)))))
```

The port is closed whether the body returns normally, raises an exception, or a continuation exits.

Re-entry Behavior

If a continuation captured inside the body is invoked from outside, the "before" thunk runs again. If a continuation from outside is invoked from inside, the "after" thunk runs:

```
> (let ((log '()) (k #f))
    (dynamic-wind
      (lambda () (set! log (cons 'in log)))
```

```
      (lambda ()
        (call/cc (lambda (c) (set! k c)))
        (set! log (cons 'body log)))
      (lambda () (set! log (cons 'out log))))
    (when (< (length log) 6) (k #f))
    (reverse log))
(in body out in body out)
```

The continuation re-enters, triggering the "in" thunk again, then the body runs again, then "out."

15.6 Practical Uses of Continuations

Implementing Exceptions

Before `guard` existed, exceptions were implemented with continuations:

```
(define (try thunk handler)
  (call/cc (lambda (exit)
    (with-exception-handler
      (lambda (e) (exit (handler e)))
      thunk))))
```

Coroutines

A simple coroutine system using continuations. It needs *two* saved continuations: `return` remembers how to jump back out to the caller, and `resume` remembers how to jump back into the coroutine:

```
(define resume #f)
(define return #f)

(define (yield value)
  (call/cc (lambda (k)
             (set! resume k)      ; where to continue later
             (return value))))    ; jump out to the caller

(define (coroutine-run proc)
  (call/cc (lambda (caller)
             (set! return caller)
             (if resume
                 (resume #f)      ; jump back in
                 (proc yield)))))  ; first run: start proc
```

```
> (define (counter yield)
    (yield "one")
    (yield "two")
    "three")
> (display (coroutine-run counter))
one
> (display (coroutine-run counter))
two
> (display (coroutine-run counter))
three
```

Each call to `coroutine-run` resumes the coroutine from where it last yielded. The first call starts `counter` and captures the caller's continuation in `return`, so `yield` can escape back out with a value; subsequent calls jump back in through `resume` and continue after the corresponding `yield`.

Non-Deterministic Choice (Backtracking)

The classic showpiece for `call/cc` is nondeterministic choice, known in the Scheme literature as the `amb` operator. The goal: `(choose 1 2 3)` returns `1`—but if the program later calls `(fail)`, control jumps *back into* that `choose`, which returns 2 instead, and everything after it replays with the new value. The program explores alternatives until one works. This is the densest code in the book; read it once, then follow the walkthrough below.

```
(define fail #f)

(define (choose . alternatives)
  (call/cc (lambda (return)
    (let loop ((alts alternatives))
      (if (null? alts)
          (fail)
          (let ((old-fail fail))
            (call/cc (lambda (k)
              (set! fail (lambda ()
                (set! fail old-fail)
                (k #f)))
              (return (car alts))))
            (loop (cdr alts))))))))
```

`choose` uses both directions of `call/cc`, just like the coroutine example. The outer capture grabs `return`—the continuation of the `(choose ...)` call itself. Invoking `(return v)` means "the value of this `choose` is `v`; continue from there." The inner capture runs once per alternative and grabs `k`—the point *inside the loop*, just after the current alternative. Invoking `(k #f)` re-enters the loop, which moves on to `(cdr alts)`.

The two are wired together through `fail`. Before handing out an alternative with `(return (car alts))`, `choose` saves the current `fail` in `old-fail` and installs a new one that does two things: restore the previous `fail`, then jump to `k` to try the next alternative. So `fail` is a chain of saved resume points—each pending `choose` contributes a link, and calling `(fail)` rewinds to the most recent choice point. When a `choose` runs out of alternatives, `fail` has already been restored to the *enclosing* choice point, so the failure propagates outward—and at the top level, to whatever `fail` was originally: below, a procedure that reports there is no solution.

Note that this genuinely requires full `call/cc`: `fail` re-enters `k` after the `choose` call has already returned, which is exactly what `call/ec` forbids.

```
> (set! fail (lambda () (error "no solution")))
> (let ((x (choose 1 2 3))
        (y (choose 4 5 6)))
    (if (= (+ x y) 8)
        (list x y)
        (fail)))
(2 6)
```

The search tries combinations until `(+ x y)` equals 8: it backtracks through `(1 4)`, `(1 5)`, `(1 6)`, `(2 4)`, `(2 5)`, then finds `(2 6)`. Follow one step of that trace: `(1 6)` fails, and `y`'s `choose` has no alternatives left, so the failure propagates to `x`'s choice point. Re-entering it makes `(choose 1 2 3)` return `2`, and the body of the `let` replays from that point—including a *fresh* call to `(choose 4 5 6)`, which starts over at `4`.

15.7 When to Use Continuations

Continuations are powerful but should be used sparingly:

- **Use `call/ec`** for early returns, loop breaks, and search. This is common and cheap.
- **Use `guard`** for exception handling. It is built on continuations but provides a cleaner interface.

- **Use `call/cc`** only for advanced patterns: coroutines, generators, backtracking, or implementing new control structures. These are rare in application code.
- **Always pair with `dynamic-wind`** when resources need cleanup across continuation invocations.

Most programs never need full `call/cc`. But knowing it exists—and that all control flow in Scheme is ultimately built on continuations—gives you a deep understanding of the language's power.

Summary

- A continuation captures "the rest of the computation" as a first-class value.
- `call/ec` captures an escape continuation for early return and breaking out of loops—efficient and common.
- `call/cc` captures a full, reusable continuation, enabling coroutines and backtracking—powerful but rarely needed.
- `dynamic-wind` runs setup and teardown code even when control jumps into or out of its body.

Exercises

Exercise 15.1: Find-First with call/ec

Use `call/ec` to write a `find-first` procedure that takes a predicate and a list, and returns the first element satisfying the predicate, or `#f` if none does. Do not use the built-in `find`—use `for-each` with an escape continuation instead.

```
> (find-first even? '(1 3 4 7 8))
4
> (find-first negative? '(1 2 3))
#f
> (find-first string? '(1 "hello" 3))
"hello"
```

Exercise 15.2: Early Exit from Nested Loops

Use `call/ec` to write a `find-pair-summing-to` procedure that takes a target number and a list, and returns the first pair of elements whose sum equals the target (as a list of two elements), or `#f` if no such pair exists. This is like Python's nested loop with an early `return`.

```
> (find-pair-summing-to 10 '(1 3 7 4 6))
(3 7)
> (find-pair-summing-to 100 '(1 2 3))
#f
```

Hint: nest two `for-each` calls inside a single `call/ec`.

Exercise 15.3: Resource Cleanup with dynamic-wind

Write a `with-resource` procedure that takes three arguments: an *acquire* thunk (returns a resource), a *body* procedure (receives the resource and uses it), and a *release* procedure (cleans up the resource). Use `dynamic-wind` to guarantee the release procedure runs even if an error is raised.

```
> (with-resource
    (lambda () (display "acquiring\n") 'my-resource)
    (lambda (r) (display "using ") (display r) (newline)
    42)
    (lambda (r) (display "releasing ") (display r)
    (newline)))
acquiring
using my-resource
releasing my-resource
42
```

Verify that the release runs even when the body raises an error.

Exercise 15.4: Challenge: Simple Generator

Implement a generator using `call/cc`. Write a `make-generator` procedure that takes a procedure. The procedure receives a `yield` function; each call to `yield` suspends the generator and returns the yielded value. Calling the generator again resumes from where it left off. When the procedure returns, the generator returns `'done` on all subsequent calls.

```
> (define gen (make-generator
    (lambda (yield)
      (yield 1)
      (yield 2)
      (yield 3))))
> (gen)
1
> (gen)
2
> (gen)
3
> (gen)
done
```

This mirrors Python's `yield` statement but is built entirely from `call/cc`.

Hint: start from the coroutine example in this chapter. You need the same two saved continuations—one to jump out to the caller, one to jump back in—but packaged inside a closure so each generator gets its own pair, plus a flag that flips to `'done` when the procedure finishes.

This completes Part III. In Part IV, we explore features beyond the R7RS standard—the SRFI ecosystem, the foreign function interface, and concurrency.

λ

Part IV

Beyond R7RS

This part explores what Kaappi adds on top of the standard: its catalog of SRFI libraries, the C foreign function interface for calling native code, and concurrency with fibers and threads. If you already know Scheme, this is a good place to start.

Chapter 16
The Standard Library

Beyond the R7RS standard libraries (covered in Chapter 12), Kaappi provides 72 SRFIs (Scheme Requests for Implementation)—community-standard libraries that extend Scheme's capabilities. This chapter surveys the most useful SRFIs, showing practical examples of each. For the complete catalog of all 72 SRFIs Kaappi supports, see Appendix C.

16.1 What Are SRFIs?

SRFIs are standardized library specifications for Scheme. Each SRFI has a number, a title, and a precise specification. They are the Scheme equivalent of Python's PEPs or JavaScript's TC39 proposals—except that each SRFI defines a concrete library you can import and use immediately.

Import any SRFI by number:

```
(import (srfi 1))   ; List library
(import (srfi 69))  ; Hash tables
(import (srfi 13))  ; String library
```

Kaappi's 72 SRFIs fall into two categories:

- **8 built-in SRFIs** — implemented in Zig for performance, always available.
- **64 portable SRFIs** — pure Scheme implementations loaded on demand from `.sld` files.

In practice the split is invisible: both kinds are imported with the same `(import (srfi N))` form. The built-in ones are simply faster.

16.2 SRFI-1: Extended List Library

SRFI-1 is the most commonly used extension. It provides dozens of list procedures beyond what `(scheme base)` offers:

```
> (import (srfi 1))
> (iota 5)
(0 1 2 3 4)
> (iota 5 1)
(1 2 3 4 5)
> (iota 5 0 2)
(0 2 4 6 8)
```

Searching and Filtering

```
> (find even? '(1 3 5 4 7))
4
> (any even? '(1 3 5 7))
#f
> (any even? '(1 3 4 7))
#t
> (every positive? '(1 2 3 4))
#t
> (partition even? '(1 2 3 4 5 6))
(2 4 6)
(1 3 5)
```

Folding and Reducing

```
> (fold + 0 '(1 2 3 4 5))
15
> (fold-right cons '() '(a b c))
(a b c)
> (reduce max 0 '(3 7 2 9 4))
9
> (unfold (lambda (x) (> x 5))
          (lambda (x) (* x x))
          (lambda (x) (+ x 1))
          1)
(1 4 9 16 25)
```

Zipping and Unzipping

```
> (zip '(a b c) '(1 2 3))
((a 1) (b 2) (c 3))
> (unzip2 '((a 1) (b 2) (c 3)))
(a b c)
(1 2 3)
```

Other Highlights

```
> (take '(a b c d e) 3)
(a b c)
> (drop '(a b c d e) 3)
(d e)
> (concatenate '((1 2) (3 4) (5 6)))
(1 2 3 4 5 6)
> (delete-duplicates '(1 2 3 2 1 4))
(1 2 3 4)
```

16.3 SRFI-13: String Library

SRFI-13 extends Scheme's string operations significantly:

```
> (import (srfi 13))
> (string-contains "hello world" "world")
6
> (string-prefix? "he" "hello")
#t
> (string-suffix? "ld" "world")
#t
> (string-trim "  hello  ")
"hello  "
> (string-trim-both "  hello  ")
"hello"
> (string-join '("one" "two" "three") ", ")
"one, two, three"
> (string-split "a,b,c,d" ",")
("a" "b" "c" "d")
```

> **Coming from Python.** `string-join` is Python's `", ".join([ ... ])`. `string-split` is Python's `"a,b,c,d".split(",")`.

One portability caveat: `string-split` is a Kaappi extension to `(srfi 13)`. The SRFI itself defines `string-tokenize` instead, so code that uses `string-split` may need adjusting on other Schemes.

16.4 SRFI-69: Hash Tables

Hash tables provide O(1) average key lookup—much faster than association lists for large collections:

```
> (import (srfi 69))
> (define ht (make-hash-table))
> (hash-table-set! ht "name" "Alice")
> (hash-table-set! ht "age" 30)
> (hash-table-ref ht "name")
"Alice"
> (hash-table-ref ht "age")
30
> (hash-table-exists? ht "email")
#f
> (hash-table-ref/default ht "email" "unknown")
"unknown"
```

> **Keys always compare with `equal?`.** Kaappi accepts SRFI-69's optional equality and hash arguments to `make-hash-table` but ignores them: every table compares keys with `equal?`. Strings, lists, and vectors therefore work as keys by value, as the examples above show. Records do not—a record is `equal?` only to itself (Chapter 13), so a record key can be retrieved only with the very same object. To index by a record's contents, use a key built from plain data, such as a list of its field values.

Iteration

```
> (hash-table-set! ht "city" "London")
> (hash-table-keys ht)
```

```
("age" "city" "name")
> (hash-table-values ht)
(30 "London" "Alice")
> (hash-table-walk ht
    (lambda (key value)
      (display key) (display ": ")
      (display value) (newline)))
name: Alice
city: London
age: 30
```

Notice that the iteration order is *unspecified*—it matches neither the insertion order nor each other across calls. Never rely on the order in which a hash table yields its entries; sort the keys first if you need a stable order.

Building from Data

To load a hash table from existing data, such as an association list, iterate over the entries:

```
> (define color-alist '(("red" . "#ff0000")
                        ("green" . "#00ff00")
                        ("blue" . "#0000ff")))
> (define ht (make-hash-table))
> (for-each (lambda (entry)
              (hash-table-set! ht (car entry) (cdr entry)))
            color-alist)
> (hash-table-ref ht "green")
"#00ff00"
```

16.5 SRFI-133: Vector Library

SRFI-133 adds higher-order operations on vectors:

```
> (import (srfi 133))
> (vector-map + #(1 2 3) #(10 20 30))
#(11 22 33)
> (vector-count even? #(1 2 3 4 5 6))
3
> (vector-any odd? #(2 4 5 6))
#t
> (vector-every positive? #(1 2 3 4))
```

```
#t
> (vector-index even? #(1 3 4 7))
2
```

16.6 SRFI-132: Sorting

```
> (import (srfi 132))
> (list-sort < '(5 3 1 4 2))
(1 2 3 4 5)
> (list-sort string<? '("banana" "apple" "cherry"))
("apple" "banana" "cherry")
> (vector-sort < #(5 3 1 4 2))
#(1 2 3 4 5)
```

The comparator is a two-argument predicate. Use `<` for ascending numbers, `>` for descending, `string<?` for strings.

16.7 SRFI-27: Random Numbers

```
> (import (srfi 27))
> (random-integer 100)
42    (your numbers will differ)
> (random-integer 100)
73
> (random-real)
0.6234981723
```

`random-integer` returns a random integer in `[0, n)`. `random-real` returns a random real in `[0, 1)`.

16.8 SRFI-115: Regular Expressions

```
> (import (srfi 115))
> (regexp-matches '(: "hello") "hello world")
#f
> (regexp-search '(: "world") "hello world")
#<<regexp-match> "hello world" #((6 11))>
> (regexp-replace '(+ num) "age: 25" "XX")
```

```
"age: XX"
> (regexp-extract '(+ alpha) "foo123bar456baz")
("foo" "bar" "baz")
```

SRFI-115 uses S-expression syntax for patterns instead of string-based regex. `(+ num)` matches one or more digits, `(+ alpha)` matches one or more letters. A match object records the searched string and the span of each match—here characters 6 to 11.

16.9 SRFI-158: Generators

Generators are lazy sequences—they produce values one at a time on demand:

```
> (import (srfi 158))
> (define g (make-range-generator 1 6))
> (g)
1
> (g)
2
> (g)
3
> (generator→list (make-range-generator 0 5))
(0 1 2 3 4)
> (generator→list
    (gfilter even? (make-range-generator 0 10)))
(0 2 4 6 8)
```

Generators are useful for processing large or infinite sequences without materializing them all in memory.

16.10 SRFI-64: Unit Testing

Every project you keep needs tests, and Scheme's standard test framework is SRFI-64. You group related checks between `test-begin` and `test-end`, and state expectations with `test-equal` (compares with `equal?`) and `test-assert` (expects a true value). Each test takes an optional descriptive name:

```
> (import (srfi 64))
> (test-begin "arith")
%%%% Starting test arith  (Writing full log to "arith.log")
> (test-equal "two plus two" 4 (+ 2 2))
> (test-assert "ten is even" (even? 10))
```

```
> (test-equal "off by one" 5 (+ 2 2))
FAIL off by one
> (test-end "arith")
# of expected passes      2
# of unexpected failures  1
```

Passing tests are silent. Failures are reported the moment they run, and `test-end` prints the tally for the group. The full record—every test's name, actual value, and expected value—is written to the log file named after the group (here `arith.log`), which is where you look when a failure message alone is not enough.

`test-error` asserts that an expression *raises* an error, which is how you test the failure paths of Chapter 14:

```
> (test-begin "errors")
%%%% Starting test errors  (Writing full log to "errors.log")
> (test-error "car of non-pair" (car 5))
> (test-end "errors")
# of expected passes      1
```

In a real project, tests live in their own files—the ecosystem convention is a `tests/` directory next to `lib/`—and each file is an ordinary script you run with `kaappi tests/test-mylib.scm`. Import the library under test, exercise its exports, and let the summary tell you where you stand. To run a whole suite at once, hand the directory to the built-in runner: `kaappi test tests/` executes each file in its own subprocess, aggregates the pass/fail tallies, and exits non-zero on any failure—exactly what a CI job wants. Kaappi can also report which exported procedures your tests never called: run the suite with `kaappi --coverage` (Appendix B).

If SRFI-64 feels heavyweight for a quick script, SRFI-78 offers a one-form alternative: `check` compares an expression against an expected value and reports immediately:

```
> (import (srfi 78))
> (check (+ 2 2) => 4)
(1) (+ 2 2) => 4 ; correct
> (check-report)

Checks: 1 total, 1 passed, 0 failed.
```

Beyond the SRFIs, the ecosystem provides `kaappi-test` (a richer test framework with suites and CI-friendly output) and `kaappi-bdd` (behavior-driven `describe/it/expect` style); see Appendix E.

16.11 Other Useful SRFIs

SRFI-17: Generalized `set!`

With SRFI-17, `set!` accepts an *accessor call* as its target and updates the place that accessor reads—the closest Scheme comes to Python's `p[0] = x`:

```
> (import (srfi 17))
> (define p (list 1 2 3))
> (set! (car p) 99)
> p
(99 2 3)
> (define v (vector 'a 'b 'c))
> (set! (vector-ref v 2) 'z)
> v
#(a b z)
```

Setters come pre-defined for the common accessors, including `car`, `cdr`, `vector-ref`, and `string-ref`—`(set! (vector-ref v 2) 'z)` means `(vector-set! v 2 'z)`. The mutation caveats of Chapter 5 still apply; this changes the spelling, not the semantics.

SRFI-61: A More General `cond`

Chapter 6's `cond` can pass a truthy test value to a receiver with ⇒. SRFI-61 extends the clause to `(generator guard ⇒ receiver)`: the generator produces a value, the *guard* decides whether the clause fires, and the receiver gets the value. That separates "produced a value" from "produced the right kind of value":

```
> (import (srfi 61))
> (define (classify s)
    (cond ((string→number s) exact-integer? =>
           (lambda (n) (list 'integer n)))
          ((string→number s) number? =>
           (lambda (n) (list 'number n)))
          (else 'not-a-number)))
> (classify "42")
(integer 42)
> (classify "3.5")
```

```
(number 3.5)
> (classify "hello")
not-a-number
```

With plain `cond`, the first clause would fire for *any* number; the `exact-integer?` guard lets `"3.5"` fall through to the second clause without parsing the string twice by hand.

SRFI-26: Partial Application with `cut`

```
> (import (srfi 26))
> (map (cut + <> 10) '(1 2 3))
(11 12 13)
> (map (cut string-append "prefix-" <>) '("a" "b" "c"))
("prefix-a" "prefix-b" "prefix-c")
```

`cut` creates a partial application. `<>` marks the argument slot.

SRFI-28: Basic Format Strings

```
> (import (srfi 28))
> (format "Hello, ~a! You are ~a years old." "Alice" 30)
"Hello, Alice! You are 30 years old."
```

SRFI-113: Sets and Bags

A set needs a *comparator* (from SRFI-128) that tells it how to compare elements; `make-default-comparator` builds one that handles all the common types:

```
> (import (srfi 113) (srfi 128) (srfi 132))
> (define s (set (make-default-comparator) 1 2 3 4 5))
> (set-contains? s 3)
#t
> (set-contains? s 9)
#f
> (list-sort < (set→list
                (set-union s (set (make-default-comparator) 4
    5 6 7))))
(1 2 3 4 5 6 7)
```

`set→list` returns the elements in no particular order, so the example sorts them for display.

SRFI-189: Maybe and Either

```
> (import (srfi 189))
> (define result (just 42))
> (maybe? result)
#t
> (maybe-ref result)
42
> (maybe-ref/default nothing 0)
0
```

`(just v)` wraps a present value; `nothing` is the singleton "no value" object.

`maybe` is like Python's `Optional` or Rust's `Option`—it represents a value that might not exist.

16.12 Beyond SRFIs: Lazy Evaluation with `(scheme lazy)`

Not everything in the standard library is a SRFI. R7RS itself ships the `(scheme lazy)` library, which lets you defer a computation until its value is actually needed. `delay` wraps an expression into a *promise* without evaluating it; `force` evaluates it on first use:

```
> (import (scheme lazy))
> (define p (delay (begin (display "computing ... ")
                          (newline)
                          42)))
> (force p)
computing ...
42
> (force p)
42
```

Note that `computing ...` printed only once: promises *memoize* their result, so the expression runs at most once no matter how often you force it. That makes `delay` a natural fit for expensive values that might never be needed—a parsed config file, a database connection, a large table.

Two companions round out the library. `make-promise` wraps an already-computed value so it can be used where a promise is expected, and `delay-force` is the tail-recursive variant of `delay` used when a promise's computation itself returns a promise:

```
> (define q (make-promise 7))
> (force q)
7
> (promise? q)
#t
```

Coming from Python. A promise is like `functools.cache` applied to a zero-argument function: the first call computes, subsequent calls return the cached result. It is lazy initialization as a first-class value.

For programming with entire lazy sequences—infinite lists that materialize on demand—see SRFI-41 (streams) in Appendix C.

16.13 Evaluating Data as Code: `(scheme eval)`

Chapter 10 showed that `read` turns text into data structures without running anything. The `(scheme eval)` library supplies the other half: `eval` takes a data structure and evaluates it as code. Together they close the loop behind the phrase "code is data"—programs are lists, and lists can become programs.

`eval` takes two arguments: the expression to evaluate and an *environment* that supplies the bindings the expression may use. Build one with `environment`, which accepts the same import sets as an `import` form:

```
> (import (scheme eval) (scheme repl))
> (eval '(+ 1 2) (environment '(scheme base)))
3
> (define form '(* 6 7))
> (eval form (environment '(scheme base)))
42
```

The expression is ordinary list data. You can assemble it with `list`, `cons`, or quasiquote (Chapter 5) before handing it over—no string pasting involved.

An environment built by `environment` contains only the named libraries' bindings; your own definitions are not visible in it. To evaluate an expression

against the live top-level environment, where everything you have defined is in scope, use `interaction-environment` from `(scheme repl)`:

```
> (define x 10)
> (eval '(+ x 1) (interaction-environment))
11
```

Combining `read` on a string port (Chapter 10) with `eval` gives the closest analogue to Python's `eval("x + 5")`:

```
> (define (eval-string s)
    (eval (read (open-input-string s))
          (interaction-environment)))
> (eval-string "(+ x 5)")
15
```

> **eval Runs Code.** `read` on untrusted input builds data at worst (Chapter 10); `eval` on untrusted input runs arbitrary code—file deletion included. Never feed `eval` anything that arrives over a network or from a user you do not control. And when the code is known as you write the program, you do not need `eval` at all: a procedure handles computed *values*, and a macro (Chapter 11) handles computed *syntax*. Reach for `eval` only when program text genuinely arrives at run time—a configuration file of formulas, a calculator, a REPL of your own.

16.14 Clocks and Timing: `(scheme time)`

R7RS also ships a minimal clock library, `(scheme time)`, with exactly three procedures. `current-second` returns the current time as a number of seconds since January 1, 1970—the same epoch as Python's `time.time()`. It is the right tool for timestamps, like the record timestamps in Chapter 13. For measuring *durations*, use `current-jiffy`: a monotonically increasing tick count in implementation-defined units, with `jiffies-per-second` giving the conversion factor. In Kaappi a jiffy is a microsecond:

```
> (import (scheme time))
> (jiffies-per-second)
1000000
```

A small helper turns any thunk into a timed run:

```
(define (measure thunk)
  (let* ((start (current-jiffy))
         (result (thunk))
         (elapsed (- (current-jiffy) start)))
    (display "elapsed: ")
    (display (inexact (/ elapsed (jiffies-per-second))))
    (display "s")
    (newline)
    result))
```

```
(measure (lambda () (length (iota 1000000))))
```

Sample run

```
elapsed: 0.021882s
```

The elapsed line differs on every run—this is a sample, not output to reproduce—and `measure` then returns the thunk's value, here `1000000`. Chapter 11 wrapped the same pattern in a `time-it` macro, so the timed expression does not need a `lambda`.

16.15 Choosing the Right Data Structure

Need	Use	SRFI	Key operation
Ordered sequence	List	1	O(n) search
Key-value map	Hash table	69	O(1) avg lookup
Sorted collection	Sorted list/vector	132	O(n log n) sort
Unique elements	Set	113	O(1) avg membership
FIFO queue	Queue	117	O(1) push/pop
Double-ended queue	Deque	134	O(1) at both ends
Indexed random access	Vector	133	O(1) index

Summary

- SRFIs are community-standard libraries; Kaappi ships 72 of them, each imported with `(import (srfi N))`.

- SRFI-1 extends list processing, SRFI-13 extends strings, and SRFI-69 provides hash tables.
- SRFI-133 (vectors), SRFI-132 (sorting), SRFI-27 (random numbers), and SRFI-115 (regular expressions) cover common needs.
- SRFI-158 provides generators, and other SRFIs add sets and bags, format strings, and the Maybe and Either types.
- SRFI-17 lets `set!` target accessor calls (`(set! (car p) v)`), and SRFI-61 adds a guard step to `cond`'s ⇒ clauses.
- SRFI-64 is the standard test framework (`test-begin`, `test-end`, `test-equal`, `test-assert`, `test-error`); `kaappi test` runs whole suites. SRFI-78's `check` is a lightweight alternative.
- R7RS's own `(scheme lazy)` library provides `delay` and `force` for memoized, on-demand computation.
- `eval` evaluates list data as code, against either a library environment built by `environment` or the live `interaction-environment`. Never use it on untrusted input.
- `(scheme time)` provides `current-second` for wall-clock timestamps and `current-jiffy` with `jiffies-per-second` for measuring durations.

Exercises

Exercise 16.1: Student Records with SRFI-1

Given a list of student records represented as association lists:

```
(define students
  '(("Alice" . 85) ("Bob" . 42) ("Carol" . 91)
    ("Dave" . 55)  ("Eve" . 73) ("Frank" . 38)))
```

Using SRFI-1 functions:

1. Use `partition` to split the students into those who passed (score ≥ 60) and those who failed.

2. Use `zip` to combine a list of names with a list of letter grades you assign.

3. Use `unfold` to generate a list of threshold boundaries `(0 10 20 ... 100)` without manually writing them out.

Exercise 16.2: Word Frequency Counter

Write a procedure `(word-frequencies str)` that takes a string of words (separated by spaces) and returns a SRFI-69 hash table mapping each word to its count. Then write `(top-words ht n)` that returns the `n` most frequent words as a sorted list of `(word . count)` pairs.

```
> (define ht (word-frequencies
    "the cat sat on the mat the cat"))
> (hash-table-ref ht "the")
3
> (hash-table-ref ht "cat")
2
> (top-words ht 2)
(("the" . 3) ("cat" . 2))
```

Hint: use `string-split` (SRFI-13) to tokenize and `list-sort` (SRFI-132) for sorting. In Python this would be `Counter(s.split())`.

Exercise 16.3: Email Validation with Regexps

Using SRFI-115, write a procedure `(parse-email str)` that checks whether a string looks like an email address and extracts the local part and domain. Return `#f` if the string is not a valid email, or a list `(local domain)` if it is.

```
> (parse-email "alice@example.com")
("alice" "example.com")
> (parse-email "not-an-email")
#f
```

Use `regexp-matches` with submatch groups. A simple pattern is: one or more alphanumeric or dot/hyphen/underscore characters, then @, then one or more alphanumeric or dot/hyphen characters. You do not need to handle every edge case of RFC 5322.

Exercise 16.4: Fibonacci Generator

Using SRFI-158, create a procedure `(make-fib-generator)` that returns a generator producing the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, ... Then write `(generator-take g n)`, which calls a generator `n` times and returns the collected values as a list.

```
> (define fibs (make-fib-generator))
> (fibs)
0
> (fibs)
1
> (fibs)
1
> (fibs)
2
> (generator-take (make-fib-generator) 10)
(0 1 1 2 3 5 8 13 21 34)
```

Hint: use `make-unfold-generator` or build one manually with a closure that tracks two state variables. In Python, this would be a function with `yield`.

Exercise 16.5: Test Suite for Your Own Code

Pick two exercises you solved earlier in this chapter (for example, `word-frequencies` and `parse-email`) and write a SRFI-64 test suite for them in a separate file. Cover the happy path, at least one edge case each (an empty string; a string with no @), and use `test-error` for an input that should raise an error. Run the file with `kaappi` and make every test pass. Then break one procedure on purpose, rerun the suite, and read the log file to see how the failure is reported.

In the next chapter, we explore calling C code from Scheme using the foreign function interface.

Chapter 17
Foreign Function Interface

Sometimes you need to use a library written in C—a graphics toolkit, a database driver, a compression algorithm, or a system API. Kaappi's Foreign Function Interface (FFI) lets you call C functions directly from Scheme without writing any C code yourself. This chapter covers opening shared libraries, binding functions, handling types, and passing Scheme callbacks into C.

17.1 The `(kaappi ffi)` Library

The FFI is provided by the `(kaappi ffi)` library:

```
(import (kaappi ffi))
```

The core procedures are:

Procedure	Purpose
`ffi-open`	Open a shared library (.dylib / .so)
`ffi-fn`	Bind a C function from the library
`ffi-close`	Close the library handle
`ffi-callback`	Create a C function pointer from a Scheme procedure
`ffi-callback-release`	Free a callback
`ffi-callback?`	Test if a value is an FFI callback
`ffi-bytevector-ptr`	Get the data pointer of a bytevector (for passing buffers to C)

17.2 Opening a Shared Library

```
> (import (kaappi ffi))
> (define libm (ffi-open "libm.dylib"))  ; macOS
```

On Linux, use the `.so` name:

```
(define libm (ffi-open "libm.so.6"))
```

The library is loaded into the process and stays open until you call `ffi-close` or the program exits.

Library names by platform. macOS uses `.dylib` extensions (`libm.dylib`, `libsqlite3.dylib`). Linux uses `.so` with version suffixes (`libm.so.6`, `libsqlite3.so.0`). You can also pass an absolute path.

17.3 Binding C Functions

`ffi-fn` looks up a symbol in the library and creates a callable Scheme procedure:

```
(ffi-fn library "function-name" '(param-types ...)
    'return-type)
```

A function can take at most 16 parameters.

Example: Math Functions

```
> (define c-sqrt (ffi-fn libm "sqrt" '(double) 'double))
> (define c-pow (ffi-fn libm "pow" '(double double) 'double))
> (define c-ceil (ffi-fn libm "ceil" '(double) 'double))
> (c-sqrt 2.0)
1.4142135623730951
> (c-pow 2.0 10.0)
1024.0
> (c-ceil 3.2)
4.0
```

The bound function behaves like any other Scheme procedure—you can pass it to `map`, store it in a variable, or use it in higher-order patterns.

Example: String Functions

```
> (define libc (ffi-open "libc.dylib"))
> (define c-strlen (ffi-fn libc "strlen" '(string) 'size_t))
> (c-strlen "hello")
5
> (c-strlen "")
0
```

17.4 Supported Types

The following C types are supported in parameter and return type positions:

FFI type	C type
`int`	`int` (32-bit signed)
`long`	`long` (64-bit on 64-bit systems)
`int8` / `uint8`	`int8_t` / `uint8_t`
`int16` / `uint16`	`int16_t` / `uint16_t`
`int32` / `uint32`	`int32_t` / `uint32_t`
`int64` / `uint64`	`int64_t` / `uint64_t`
`size_t`	`size_t`
`float`	`float` (32-bit)
`double`	`double` (64-bit)
`char`	`char`
`bool`	`_Bool`
`string`	`const char*` (null-terminated)
`pointer`	`void*` (opaque pointer)
`void`	`void` (return type only)

Type Conversions

- **Strings:** Scheme strings are automatically converted to null-terminated C strings when passed as arguments, and C strings are converted back to Scheme strings when returned.
- **Numbers:** Scheme integers and floats are converted to the appropriate C numeric type. Out-of-range values are rejected with an error naming the FFI function and the expected type.
- **Characters:** The `char` type takes and returns Scheme characters—a C function declared `(char) → char` maps `#\a` to `#\A`, not `97` to `65`.

- **Pointers:** Opaque pointer values are represented as Scheme FFI pointer objects. You cannot dereference them directly from Scheme.
- **Void:** Use `'void` as the return type for C functions that return nothing.

17.5 A Complete Example: System Information

Here is a more realistic example that combines several libc functions to query system information:

```
(import (kaappi ffi))

(define libc (ffi-open "libc.dylib"))

;; Bind functions we need
(define c-getenv
  (ffi-fn libc "getenv" '(string) 'string))
(define c-strlen
  (ffi-fn libc "strlen" '(string) 'size_t))
(define c-time
  (ffi-fn libc "time" '(pointer) 'long))
(define c-getpid
  (ffi-fn libc "getpid" '() 'int))
```

```
> (c-getenv "HOME")
"/Users/alice"    (your values will differ)
> (c-getenv "SHELL")
"/bin/zsh"
> (c-strlen (c-getenv "HOME"))
12
> (c-getpid)
42137
> (c-time 0)
1751234567
```

```
;; Build a system report
(define (system-info)
  (list
    (cons "home" (c-getenv "HOME"))
    (cons "shell" (c-getenv "SHELL"))
    (cons "pid" (c-getpid))
    (cons "time" (c-time 0))))
```

```
;; Clean up
(ffi-close libc)
```

Why not SQLite?. The SQLite C API uses pointer-to-pointer out-parameters (`sqlite3**`) to return database handles, which requires low-level pointer manipulation beyond what the FFI provides directly. In practice, you would use the `kaappi-sqlite` package (`thottam install kaappi-sqlite`), which wraps these details in a clean Scheme interface.

17.6 Passing Buffers with Bytevectors

Many C functions do not return a string—they fill a buffer you hand them. `ffi-bytevector-ptr` returns the address of a bytevector's data, which you can pass wherever C expects a pointer. Here `getcwd` writes the current directory into a bytevector as a 0-terminated C string:

```
(import (kaappi ffi))

(define libc (ffi-open "libc.dylib"))
(define c-getcwd
  (ffi-fn libc "getcwd" '(pointer size_t) 'pointer))

;; Extract the C string that ends at the first 0 byte
(define (bytevector-cstring bv)
  (let loop ((i 0))
    (if (zero? (bytevector-u8-ref bv i))
        (utf8→string (bytevector-copy bv 0 i))
        (loop (+ i 1)))))

(define buf (make-bytevector 4096 0))
(define result (c-getcwd (ffi-bytevector-ptr buf) 4096))
(display (bytevector-cstring buf))
```

```
/Users/alice/projects
```

Keep the bytevector reachable for as long as C holds the pointer. Kaappi's garbage collector does not move objects, so the address stays valid while the bytevector is alive—but if the bytevector is collected, the pointer dangles.

17.7 Callbacks: Passing Scheme to C

Some C APIs expect function pointers as arguments (e.g., comparators for `qsort`, event handlers, iteration callbacks). `ffi-callback` creates a C-compatible function pointer from a Scheme procedure:

```
(define cb (ffi-callback scheme-proc '(param-types ...)
    'return-type))
```

Callbacks support only a fixed set of signatures:

Parameters	Return type
`()`	`void`
`(int)`	`void`
`(pointer)`	`void`
`(pointer)`	`int`
`(int pointer)`	`int`
`(pointer pointer)`	`void`
`(pointer pointer)`	`int`

Any other combination raises `unsupported callback signature`. At most 32 callbacks can exist at once, which is another reason to release them promptly.

Example: Sorting Bytes with `qsort`

Pointer arguments arrive in the Scheme procedure as raw addresses. You cannot dereference them from Scheme—but you can hand them straight back to C. This comparator sorts the bytes of a bytevector with C's `qsort` by delegating each comparison to `memcmp`:

```
(define c-qsort
  (ffi-fn libc "qsort"
    '(pointer size_t size_t pointer) 'void))
(define c-memcmp
  (ffi-fn libc "memcmp" '(pointer pointer size_t) 'int))

;; qsort passes addresses of two elements; compare 1 byte each
```

```
(define byte-compare
  (ffi-callback
    (lambda (a b) (c-memcmp a b 1))
    '(pointer pointer)
    'int))

(define data (bytevector 3 1 4 1 5 9 2 6))
(c-qsort (ffi-bytevector-ptr data) 8 1 byte-compare)
(ffi-callback-release byte-compare)
(display data)
```

```
#u8(1 1 2 3 4 5 6 9)
```

> **Always release callbacks.** Callbacks pin Scheme closures so the garbage collector cannot reclaim them. Always call `ffi-callback-release` when the callback is no longer needed to prevent memory leaks.

If the Scheme procedure raises an error while C is calling it, the error is not lost: it is re-raised in your program as soon as the C call that invoked the callback returns, where `guard` can catch it as usual. The C code itself never sees a Scheme exception—it runs to completion first.

17.8 Closing Libraries

When you are done with a library, close it to release the handle:

```
> (ffi-close libm)
```

After closing, any functions bound from that library become invalid. In practice, most programs open libraries at startup and never close them (the OS reclaims everything on exit).

17.9 Error Handling

FFI calls can crash your program if you get the types wrong—there is no safety net between Scheme and C. Common pitfalls:

- **Wrong types:** Passing an integer where a pointer is expected (or vice versa) causes undefined behavior.
- **Null pointers:** C functions may return NULL; check pointer values before using them.
- **Lifetime issues:** If you pass a Scheme string to C and the C code stores the pointer for later use, the Scheme GC may collect the string. Copy the data on the C side if it needs to persist.
- **Thread safety:** Do not call FFI functions from multiple Kaappi fibers simultaneously unless the C library is thread-safe.

> **FFI bypasses safety.** The FFI is inherently unsafe. Incorrect type declarations, buffer overflows, and use-after-free in C code can crash the Kaappi process. Test FFI bindings carefully and prefer ecosystem packages (`kaappi-sqlite`, `kaappi-pg`, etc.) when available.

The FFI error messages you are most likely to encounter, with their fixes, are listed in Appendix D.

Debugging FFI Problems

When FFI calls misbehave, use this checklist:

1. **Verify the library loaded.** If `ffi-open` returns without error, the library is found. If it raises an error, check the file name and path. On macOS, use `otool -L` to inspect a binary's dependencies; on Linux, use `ldd`.
2. **Check the symbol name.** C++ name-mangling, static linking, and version suffixes can cause `ffi-fn` to fail. Use `nm -gU libfoo.dylib | grep function_name` to confirm the symbol exists.
3. **Match types exactly.** A function declared as returning `int` but actually returning `size_t` may work on some platforms and silently corrupt values on others. Consult the C header file.
4. **Test in isolation.** Bind one function at a time in the REPL. Call it with known inputs and compare against expected outputs before integrating into a larger program.

17.10 FFI and Sandbox Mode

Kaappi's `--sandbox` mode disables the FFI entirely. This is useful for running untrusted code:

```
kaappi --sandbox untrusted.scm
```

In sandbox mode, `(import (kaappi ffi))` itself fails with `sandbox: cannot load library from file`, and none of the FFI procedures are defined. See Appendix B for the full set of command-line options, including `--sandbox`.

17.11 When to Use the FFI

- **Use it** when you need a C library that has no Scheme wrapper (niche libraries, system APIs, hardware interfaces).
- **Prefer ecosystem packages** (`thottam install ...`) when they exist—they handle error checking, memory management, and provide idiomatic Scheme interfaces.
- **Consider writing a library** if you find yourself wrapping more than a few functions. Publish it via thottam so others benefit.

Summary

- The `(kaappi ffi)` library calls C functions in shared libraries at runtime.
- `ffi-open` loads a shared library and `ffi-fn` binds a C function to a Scheme procedure given its type signature.
- The FFI supports integer, floating-point, string, and pointer types; bytevectors pass buffers via `ffi-bytevector-ptr`, and Scheme procedures become C function pointers via `ffi-callback` (fixed signature set, released with `ffi-callback-release`).
- The FFI is inherently unsafe and is disabled by `--sandbox`; prefer ecosystem packages when they exist.

Exercises

Exercise 17.1: Binding `strlen`

Use the FFI to bind `strlen` from the C standard library. Open the appropriate shared library for your platform (`libc.dylib` on macOS, `libc.so.6` on Linux), bind `strlen` with the correct type signature, and test it:

```
> (c-strlen "hello")
5
> (c-strlen "")
0
> (c-strlen "Kaappi Scheme")
13
```

What happens if you declare the return type as `'int` instead of `'size_t`? Does it still work for short strings? Think about why `size_t` is the correct choice (hint: consider strings longer than 2^{31} bytes).

Exercise 17.2: Comparing FFI `pow` with `expt`

Bind the C function `pow` from `libm` using the FFI and write a procedure that compares its results with Scheme's built-in `expt`:

```
(define (compare-pow base exp)
  (let ((c-result (c-pow (exact→inexact base)
                         (exact→inexact exp)))
        (s-result (expt base exp)))
    (list c-result s-result (- c-result s-result))))
```

Test with several inputs: `(compare-pow 2 10)`, `(compare-pow 3.0 3.0)`, and `(compare-pow 2 0.5)`. When do the results differ? Why might the C version and the Scheme version give slightly different answers? (Hint: think about exact vs. inexact arithmetic.)

Exercise 17.3: Safe Library Wrapper

Write a procedure `(with-library name proc)` that opens a shared library, passes the handle to `proc`, and guarantees that `ffi-close` is called even if `proc` raises an exception. This is the FFI equivalent of Python's `with open( ... ) as f:` pattern.

```
(define (with-library name proc)
  ;; Your implementation here.
  ;; Use guard or dynamic-wind to ensure cleanup.
  ... )
```

`guard` (Chapter 14) is sufficient here; `dynamic-wind` (Chapter 15) is the more general tool for paired acquire/release. Test it by binding and calling `ceil` from libm:

```
> (with-library "libm.dylib"
    (lambda (lib)
      (let ((c-ceil (ffi-fn lib "ceil"
                      '(double) 'double)))
        (c-ceil 3.2))))
4.0
```

Verify that the library is closed after the call returns. What happens if you try to use a function bound from a closed library?

In the next chapter, we cover concurrency—fibers, channels, and threads.

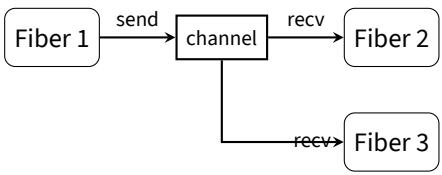
Chapter 18
Concurrency

Modern programs often need to do multiple things at once: serve multiple network connections, process data while waiting for I/O, or distribute work across CPU cores. Kaappi provides two concurrency models: *fibers* (lightweight green threads) for cooperative multitasking and *SRFI-18 threads* (OS threads) for true parallelism.

18.1 Two Models of Concurrency

Aspect	Fibers	OS Threads (SRFI-18)
Library	`(kaappi fibers)`	`(srfi 18)`
Scheduling	Cooperative (yield)	Preemptive (OS)
Parallelism	No (single OS thread)	Yes (multiple cores)
Communication	Channels	Channels + deep-copy
Overhead	Low (kilobytes per fiber)	High (VM + GC per thread)
Shared state	Same heap	Separate heaps (safe)
Use case	I/O, event loops	CPU-bound parallelism

For most concurrent programs, fibers are the right choice. Use OS threads only when you need to exploit multiple CPU cores for compute-heavy work.



18.2 Fibers

Fibers are lightweight green threads that run cooperatively within a single OS thread. They are created with `spawn`:

```
> (import (kaappi fibers))
> (define f (spawn (lambda () (* 6 7))))
> (fiber-join f)
42
```

`spawn` takes a zero-argument procedure (a thunk), starts it as a new fiber, and returns a fiber object. `fiber-join` blocks until the fiber completes and returns its result.

Cooperative Scheduling

Fibers run until they voluntarily yield control. Call `yield` to let other fibers run:

```
> (define (counting-fiber name n)
    (spawn (lambda ()
      (let loop ((i 0))
        (when (< i n)
          (display name) (display ":")
          (display i) (display " ")
          (yield)
          (loop (+ i 1)))))))
> (define f1 (counting-fiber "A" 3))
> (define f2 (counting-fiber "B" 3))
> (fiber-join f1)
A:0 B:0 A:1 B:1 A:2 B:2
> (fiber-join f2)
```

The fibers interleave at each `yield` point. Note that `spawn` only queues a fiber—nothing runs until the main fiber gives up control. The first `fiber-join` does exactly that, so both fibers run to completion there, and the second join returns immediately.

The flip side of cooperative scheduling is *starvation*. Here `slow` grinds through a long computation with no `yield` in sight, while `quick` only wants to print one line:

```
> (define (busy-work n)
    (if (zero? n) 'done (busy-work (- n 1))))
> (define slow (spawn (lambda ()
```

```
    (busy-work 1000000)
    (display "slow finished") (newline))))
> (define quick (spawn (lambda ()
    (display "quick finished") (newline))))
> (fiber-join quick)
slow finished
quick finished
```

We asked for `quick`'s result, but `slow` was queued first and never gave up control, so `quick` had to wait out the entire computation. A `yield` in the loop restores responsiveness:

```
> (define (busy-work n)
    (if (zero? n)
        'done
        (begin (yield) (busy-work (- n 1)))))
> (define slow (spawn (lambda ()
    (busy-work 1000000)
    (display "slow finished") (newline))))
> (define quick (spawn (lambda ()
    (display "quick finished") (newline))))
> (fiber-join quick)
quick finished
> (fiber-join slow)
slow finished
```

Now `slow` offers control back on every iteration, so `quick` runs at the first opportunity while the long computation finishes afterward. Be clear about what `yield` buys here: *responsiveness*, not *fairness*. It lets other fibers run; it does not promise an even split of processor time (the fan-out example in Section 18.3 returns to this).

> **Long computations without yield.** Since fibers are cooperative, a fiber that runs a long computation without yielding will block all other fibers. Insert `(yield)` in long loops to keep the system responsive.

Fiber Results and Errors

`fiber-join` returns the fiber's result value. If the fiber raised an exception, `fiber-join` re-raises it in the calling context:

```
> (define f (spawn (lambda () (error "oops"))))
> (guard (e (#t (error-object-message e)))
    (fiber-join f))
"oops"
```

18.3 Channels

Channels provide safe communication between fibers. A channel is a queue: `channel-send` appends a value and `channel-receive` takes the oldest one, letting other fibers run until a value is available. By default the queue is unbounded and a send returns immediately; `(make-channel n)` caps it at `n` queued values, and a send to a full bounded channel blocks the sending fiber until a receive frees a slot.

> **Coming from Go.** Go's unbuffered channels synchronize sender and receiver: a send waits until a receive happens. Kaappi's default channel is closer to an unbounded buffered channel or an Erlang mailbox—producers never wait, so one that outruns its consumer grows the queue (and memory) with the backlog. A bounded channel `(make-channel n)` behaves like Go's `make(chan T, n)`: once `n` values are queued, senders wait—backpressure that keeps the producer honest. There is no zero-capacity rendezvous.

```
> (import (kaappi fibers))
> (define ch (make-channel))
> (define sender
    (spawn (lambda ()
      (channel-send ch "hello from fiber"))))
> (channel-receive ch)
"hello from fiber"
```

Bounded Channels: Backpressure

Give `make-channel` a capacity and the channel pushes back on a producer that gets ahead. Here the channel holds at most one value, so the producer's second send cannot complete until the consumer has taken the first:

```
> (define ch (make-channel 1))
```

```
> (define producer
    (spawn (lambda ()
      (channel-send ch 'a)
      (display "sent a") (newline)
      (channel-send ch 'b)
      (display "sent b") (newline))))
> (channel-receive ch)
sent a
a
> (channel-receive ch)
sent b
b
> (fiber-join producer)
```

Follow the interleaving. The producer does not run until the first `channel-receive` blocks and hands over the scheduler; it sends `'a` into the free slot, prints, then blocks trying to send `'b` into the now-full channel. Each receive frees a slot and lets the producer take exactly one more step. That lock-step rhythm is the point: a bounded channel converts "producer floods memory" into "producer waits its turn."

Timeouts and Closing

Both `channel-receive` and `channel-send` accept an optional timeout in seconds, plus a value to return if it expires instead of raising an error:

```
> (channel-receive (make-channel) 0.05 'nothing)
nothing
```

When a stream of values ends, close the channel. `channel-close!` marks it finished: receivers drain whatever is still queued and then get an EOF object—the same value a port returns at end of input (Chapter 10)—while further sends raise an error. `channel-closed?` tests for it.

```
> (define ch (make-channel))
> (channel-send ch 10)
> (channel-close! ch)
> (channel-receive ch)
10
> (channel-receive ch)
#<eof>
> (channel-send ch 99)
error[KP3000]: channel-send: send on closed channel
```

One more safety net: if a receive (or a `fiber-join`) can never be satisfied because every other fiber is blocked or finished, Kaappi does not hang. It raises a catchable error naming the deadlock. You will meet it the first time you forget to close a channel or send a sentinel—read it as a precise bug report, not a nuisance.

Producer-Consumer Pattern

```
(import (kaappi fibers))

(define ch (make-channel))

;; Producer: sends numbers 0-9
(define producer
  (spawn (lambda ()
    (let loop ((i 0))
      (when (< i 10)
        (channel-send ch i)
        (loop (+ i 1))))
    (channel-send ch 'done))))

;; Consumer: receives until 'done
(let loop ()
  (let ((msg (channel-receive ch)))
    (unless (eq? msg 'done)
      (display (* msg msg))
      (display " ")
      (loop))))
```

```
0 1 4 9 16 25 36 49 64 81
```

Fan-Out: Multiple Workers

```
(import (kaappi fibers))

(define work-ch (make-channel))
(define result-ch (make-channel))

;; Spawn 3 worker fibers
(define workers
  (map (lambda (id)
```

```
    (spawn (lambda ()
      (let loop ()
        (let ((task (channel-receive work-ch)))
          (unless (eq? task 'stop)
            (channel-send result-ch
              (list id (* task task)))
            (loop)))))))
    '(1 2 3)))

;; Send work, then one stop sentinel per worker
(for-each (lambda (n) (channel-send work-ch n))
          '(10 20 30 40 50 60))
(for-each (lambda (w) (channel-send work-ch 'stop))
          workers)

;; Collect results, then wait for the workers to exit
(define results
  (let loop ((n 6) (acc '()))
    (if (= n 0)
        acc
        (loop (- n 1)
              (cons (channel-receive result-ch) acc)))))
(for-each fiber-join workers)
(display results)
```

```
((1 3600) (1 2500) (1 1600) (1 900) (1 400) (1 100))
```

Look at the worker ids: every result came from worker 1. With cooperative scheduling and non-blocking sends, the first worker to run keeps going until it blocks—and since the whole queue is already buffered, it drains everything before workers 2 and 3 get a turn. Fan-out pays off when each task genuinely blocks (waiting on a socket, a timer, or another channel), which hands the queue to the next worker. Always send one stop sentinel per worker: a worker that never receives one stays blocked on the channel forever.

You might hope that an explicit `(yield)` inside the worker loop would spread the work evenly. Try it: depending on where you place the call, worker 1 still takes every task, or at best hands one task each to the other workers before draining the rest. The scheduler promises only that *some* runnable fiber runs next—it makes no fairness guarantee, and `yield` is a hint, not a turnstile. If your tasks are pure computation and you need them spread across workers, fibers are the wrong tool; use OS threads (next section).

Racing Fibers

You can race multiple fibers by having them send to a shared channel. The first result wins:

```
(define (race thunk1 thunk2)
  (let ((ch (make-channel)))
    (spawn (lambda () (channel-send ch (thunk1))))
    (spawn (lambda () (channel-send ch (thunk2))))
    (channel-receive ch)))
```

```
> (race (lambda () (+ 1 2))
        (lambda () (* 3 4)))
3
```

The result is whichever fiber completes first. The other fiber's value is left in the channel (a second `channel-receive` would retrieve it). Note that with cooperative scheduling, two thunks that never block run to completion in spawn order—the first one always wins. Racing becomes meaningful when the thunks block on I/O, timers, or other channels.

18.4 OS Threads (SRFI-18)

For true parallelism across CPU cores, use SRFI-18 OS threads. Each thread gets its own VM and garbage collector with an independent heap.

> **Heavyweight concurrency.** Each OS thread creates an independent VM and garbage collector. This makes them safe (no shared mutable state) but expensive. Prefer fibers for I/O concurrency; reserve threads for CPU-bound work that benefits from multiple cores. OS threads are not available in sandbox mode or WASM builds.

```
(import (srfi 18))

(define t (thread-start!
  (make-thread
    (lambda () (* 6 7)))))

(thread-join! t)  ;=> 42
```

Creating and Starting Threads

```
(import (srfi 18))

(define (expensive-computation n)
  (let loop ((i 0) (acc 0))
    (if (= i n) acc
        (loop (+ i 1) (+ acc i)))))

(define t1 (thread-start!
  (make-thread
    (lambda () (expensive-computation 10000000))
    "worker-1")))

(define t2 (thread-start!
  (make-thread
    (lambda () (expensive-computation 20000000))
    "worker-2")))

(define result1 (thread-join! t1))
(define result2 (thread-join! t2))
```

Both computations run in parallel on separate cores.

Deep-Copy Semantics

Values crossing thread boundaries are *deep-copied*:

- At `thread-start!`: the thunk closure (and all values it captures) is deep-copied from the parent thread to the child.
- At `thread-join!`: the return value is deep-copied from the child thread back to the parent.

This means threads cannot share mutable state—there are no race conditions, no data corruption, and no need for locks in most code. The one sanctioned shared object is a channel, covered next.

> **Coming from Python.** Python's `multiprocessing` module works similarly: separate memory spaces, data copied between processes. Kaappi's threads are lighter (same process, shared code) but have the same safety guarantee of no shared mutable state.

Channels Across Threads

Waiting for `thread-join!` is not the only way to hear back from a thread. A channel handed to the thread's thunk is *promoted* to a shared, thread-safe channel, and both sides talk through it while the thread runs—results can stream back long before the join:

```
(import (kaappi fibers) (srfi 18))

(define (start-worker out)
  (thread-start!
    (make-thread
      (lambda ()
        (channel-send out (* 6 7))))))

(define ch (make-channel))
(define t (start-worker ch))
(channel-receive ch)  ;=> 42
(thread-join! t)
```

Each value sent across still crosses by copy, so the no-shared-mutable-state guarantee survives; the channel itself is the only object both threads touch. Everything from the previous sections works across threads too: capacities give you backpressure between threads, and `channel-close!` signals end-of-stream.

> **Hand the channel over explicitly.** A thread may only use a channel that was *handed* to it—captured by the thunk as a parameter or a `let`-bound local, as `out` is above. Reaching a channel through a global variable from inside the thunk does not promote it: the child's send fails and the parent's receive deadlocks. When a thread needs a channel, pass it in.

Mutexes and Condition Variables

SRFI-18 provides mutexes and condition variables for the rare cases where you need synchronization:

```
(import (srfi 18))

(define mtx (make-mutex))

(mutex-lock! mtx)
;; ... critical section ...
(mutex-unlock! mtx)
```

In practice, the deep-copy model means you rarely need mutexes. Use them only if you are integrating with external resources (files, network sockets) that require coordinated access.

18.5 Choosing Between Fibers and Threads

Scenario	Use	Why
Multiple HTTP connections	Fibers	I/O-bound, many tasks
Many connections across cores	Threads × fibers	`http-listen-parallel`
Event loop / GUI	Fibers	Cooperative scheduling
Pipeline processing	Fibers + channels	Natural producer-consumer
CPU-heavy computation	OS threads	True parallelism
Parallel map over data	Pooled threads	`parallel-map`
Simple background task	Fibers	Lower overhead

Why fibers suit I/O-bound work. A blocking-looking call like `(read-line port)` only suspends the fiber that made it, not the whole program. Kaappi runs a per-OS-thread I/O reactor: when a read or write would block, the calling fiber parks and the reactor waits on the underlying file descriptor, waking the fiber once data is ready. `thread-sleep!` gets the same treatment, parking on a timer instead of blocking the thread. This is why a fiber-based server can hold thousands of connections open on a single OS thread and GC heap—a slow connection parks its own fiber without freezing the rest. The ecosystem's `http-listen-fiber` (Appendix E) is a fiber-per-connection HTTP server built on exactly this. To use every core rather than one, `http-listen-parallel` runs one such

fiber server per OS thread; on Linux the kernel spreads incoming connections across the threads' `SO_REUSEPORT` sockets, so the machine serves *threads* × *fibers*—cores times thousands of connections. (macOS does not balance `SO_REUSEPORT`, so parallel serving there falls back to one acceptor thread feeding a worker pool over a shared channel; each worker still fiber-multiplexes its connections, so macOS reaches the same *threads* × *fibers* model with a userspace distributor. Linux stays the faster path, but both scale across cores.)

18.6 A Complete Example: Parallel Map

Here is a function that maps a procedure over a list using OS threads for parallelism:

```
(import (srfi 18))

(define (parallel-map f lst)
  (let ((threads
         (map (lambda (item)
                (thread-start!
                  (make-thread
                    (lambda () (f item)))))
              lst)))
    (map thread-join! threads)))
```

```
> (parallel-map (lambda (n) (* n n)) '(1 2 3 4 5))
(1 4 9 16 25)
```

Each element is processed on a separate thread. For CPU-intensive `f`, this achieves near-linear speedup on multi-core machines.

18.7 The `(kaappi parallel)` Library

The pattern above—one thread per element, join in order—is common enough that Kaappi ships a library for it. `(kaappi parallel)` provides `parallel-map` and `parallel-for-each`, built on a fixed pool of worker threads draining a shared task channel:

```
> (import (kaappi parallel))
> (parallel-map (lambda (n) (* n n)) '(1 2 3 4 5))
```

```
(1 4 9 16 25)
```

For finer control, build the pool yourself. `(make-pool n)` starts `n` worker threads; `pool-submit` hands one a thunk and immediately returns a task handle; `task-wait` collects that task's result; `pool-shutdown!` stops the workers:

```
> (define pool (make-pool 2))
> (define task
    (pool-submit pool (lambda () (+ 40 2))))
> (task-wait task)
42
> (pool-shutdown! pool)
```

`(processor-count)` reports how many CPU cores the machine has—the natural pool size. Under `–sandbox` and on WASM, where OS threads do not exist, the same API degrades gracefully to fiber workers: your program still runs, just without the parallelism.

One task per element has a cost. `parallel-map` submits one pool task per list element, so every element pays a cross-thread round trip; it suits lists up to a few hundred elements whose per-element work is substantial. For bigger inputs, chunk: submit one `pool-submit` task per core, each covering a slice of the input with an ordinary sequential loop. The tasks-as-envelopes model is the same one `thread-start!` uses, so thunks and results cross by copy here too.

18.8 A Complete Example: Fiber Pipeline

A pipeline where data flows through multiple stages via channels. The producer and the transformer each run as a fiber; the main fiber acts as the final stage. An `'eof` sentinel flows through the channels to shut the pipeline down:

```
(import (kaappi fibers))

(define nums (make-channel))
(define squares (make-channel))

;; Stage 1: produce 1..5
(define producer
  (spawn (lambda ()
```

```
    (for-each (lambda (n) (channel-send nums n))
              '(1 2 3 4 5))
    (channel-send nums 'eof))))

;; Stage 2: square each value
(define squarer
  (spawn (lambda ()
    (let process ()
      (let ((val (channel-receive nums)))
        (if (eq? val 'eof)
            (channel-send squares 'eof)
            (begin
              (channel-send squares (* val val))
              (process))))))))
```

```
;; Stage 3: add 1 and print (the main fiber)
(let loop ()
  (let ((val (channel-receive squares)))
    (unless (eq? val 'eof)
      (display (+ val 1)) (display " ")
      (loop))))
```

```
2 5 10 17 26
```

Each number is squared by one fiber and incremented by the next stage, with the channels handing values downstream one at a time.

The `'eof` sentinel is the hand-rolled version of what `channel-close!` provides: close a channel after the last send, and once the queue drains every receive returns an EOF object. The same producer, with the consumer testing `eof-object?` instead of comparing against a sentinel:

```
(define nums2 (make-channel))

(define producer2
  (spawn (lambda ()
    (for-each (lambda (n) (channel-send nums2 n))
              '(1 2 3 4 5))
    (channel-close! nums2))))

(let loop ()
```

```
  (let ((val (channel-receive nums2)))
    (unless (eof-object? val)
      (display (* val val)) (display " ")
      (loop))))
```

```
1 4 9 16 25
```

Closing scales better than a sentinel when a channel has several readers: every blocked receiver wakes and sees EOF, whereas a sentinel is consumed by whichever single reader happens to take it (the fan-out example sent one `'stop` per worker for exactly that reason).

Summary

- Kaappi offers two concurrency models: lightweight cooperative fibers and preemptive OS threads.
- Fibers are scheduled cooperatively and communicate through channels—unbounded by default, bounded for backpressure—supporting producer-consumer, fan-out, and racing patterns.
- Channel sends and receives take optional timeouts; `channel-close!` signals end-of-stream (receivers drain, then get an EOF object); an unsatisfiable wait raises a deadlock error instead of hanging.
- OS threads (SRFI-18) use deep-copy semantics and synchronize with mutexes and condition variables; a channel handed to a thread's thunk is promoted and shared, so threads can stream results back before the join.
- `(kaappi parallel)` adds `parallel-map` and worker pools (`make-pool`, `pool-submit`, `task-wait`) over pooled OS threads.
- Use fibers for I/O-bound concurrency and threads for CPU-bound parallelism.

Exercises

> **Exercise 18.1: Three-Stage Fiber Pipeline**

Write a three-stage fiber pipeline connected by channels: a *producer* that sends the integers 1 through 10, a *transformer* that squares each value, and a *consumer* that collects the results into a list. Use a sentinel value `'done` to signal the end of the stream.

```
(import (kaappi fibers))

(define ch1 (make-channel))  ; producer → transformer
(define ch2 (make-channel))  ; transformer → consumer

;; Producer: sends 1..10 then 'done
(spawn (lambda () ...))

;; Transformer: reads from ch1, squares, sends to ch2
(spawn (lambda () ...))

;; Consumer: reads from ch2 until 'done
;; Should produce: (1 4 9 16 25 36 49 64 81 100)
```

This is the Scheme equivalent of a Unix pipeline like `seq 10 | awk '{print $1*$1}'`.

Exercise 18.2: Fiber-Based Parallel Map

Implement `(fiber-parallel-map f lst)` using fibers and channels. Spawn one fiber per element, have each fiber send its result to a shared results channel, then collect all results in order.

```
> (fiber-parallel-map
    (lambda (x) (* x x))
    '(1 2 3 4 5))
(1 4 9 16 25)
```

Hint: to preserve order, have each fiber send a pair `(index . result)` and sort by index after collecting. Alternatively, use one channel per fiber. How does this differ from the OS-thread `parallel-map` shown earlier in the chapter? (Think about whether fibers actually compute in parallel.)

Exercise 18.3: Collecting Results from N Fibers

Write a procedure `(spawn-collect thunks)` that takes a list of zero-argument procedures, spawns each as a fiber, and returns a list of all their results. Use a single shared channel for collection.

```
> (spawn-collect
    (list (lambda () (+ 1 2))
          (lambda () (* 3 4))
          (lambda () (string-length "hello"))))
(3 12 5)
```

The results may arrive in any order—your procedure should still return them in the same order as the input thunks. This pattern is common when you need to fan-out work and wait for all results, similar to Python's `asyncio.gather()`.

Exercise 18.4: Worker Pool (Challenge)

Implement a worker pool with a fixed number of worker fibers that process jobs from a shared channel:

```
(define (make-worker-pool num-workers job-proc)
  ;; Returns two values: a job channel and a
  ;; results channel. Spawns num-workers fibers
  ;; that each loop reading from the job channel,
  ;; applying job-proc, and sending the result to
  ;; the results channel.
  ... )
```

Have each worker exit its loop when it receives `'stop`, and send one `'stop` per worker once all jobs are submitted. Test the pool with 3 workers and 10 jobs (the harness needs `(srfi 132)` for `list-sort`):

```
(let-values (((jobs results)
              (make-worker-pool 3
                (lambda (n) (* n n)))))
  ;; Submit 10 jobs, then stop the workers
  (for-each (lambda (n) (channel-send jobs n))
```

```
          '(1 2 3 4 5 6 7 8 9 10))
(for-each (lambda (i) (channel-send jobs 'stop))
          '(1 2 3))
;; Collect 10 results
(let loop ((n 10) (acc '()))
  (if (= n 0) (list-sort < acc)
      (loop (- n 1)
            (cons (channel-receive results)
                  acc)))))
```

Why would you use a fixed pool of workers instead of spawning one fiber per job? (Hint: think about what happens with 10,000 jobs that each open a network connection.) This is analogous to Python's `concurrent.futures.ThreadPoolExecutor`—and to `make-pool` from `(kaappi parallel)`, which applies the same design across OS threads.

This completes Part IV. The appendices that follow provide quick references for the R7RS standard and the Kaappi CLI.

Part V

Appendices

The appendices are reference material to reach for as you read or keep handy while you program: a quick R7RS reference, the command-line interface, the full SRFI catalog, common error messages, the ecosystem packages, a glossary, and pointers for further study.

Appendix A
R7RS Quick Reference

This appendix provides a concise reference to the R7RS-small standard. Use it as a lookup when you need to recall a syntax form, a procedure signature, or which library something belongs to. Each table names the library that provides its entries; library names abbreviate `(scheme ...)`, so `char` means `(scheme char)`.

A.1 Standard Libraries

R7RS-small partitions the standard into small libraries. Kaappi provides all of them:

Library	Contents
`(scheme base)`	Core syntax, numbers, lists, strings, vectors, bytevectors, control flow, basic I/O
`(scheme case-lambda)`	`case-lambda`
`(scheme char)`	Character classification and case conversion
`(scheme complex)`	Complex number accessors
`(scheme cxr)`	Compound pair accessors `caar` ... `cddddr`
`(scheme eval)`	`eval` and environments
`(scheme file)`	File ports and file operations
`(scheme inexact)`	Floating-point functions (`sqrt`, `sin`, `log`, ...)
`(scheme lazy)`	Promises: `delay`, `force`
`(scheme load)`	`load`
`(scheme process-context)`	Command line, environment variables, `exit`
`(scheme r5rs)`	The R5RS language in one import, for legacy code; also `scheme-report-environment` and `null-environment`
`(scheme read)`	`read`
`(scheme repl)`	`interaction-environment`
`(scheme time)`	Clock access: `current-second`, `current-jiffy`
`(scheme write)`	`write`, `display`

Imports Are Optional in Kaappi. Every built-in procedure is available in Kaappi without importing anything—`(import (scheme write))` is never required to call `display`. Write the imports anyway: they document your dependencies and keep your code portable to other R7RS implementations, where the libraries above must be imported explicitly. The 64 portable SRFIs are the exception: they load from `.sld` files only when imported (Appendix C).

A.2 Syntax Forms

All forms below are in `(scheme base)` except `case-lambda`, which lives in `(scheme case-lambda)`, and `delay`/`delay-force`, which live in `(scheme lazy)`.

Definitions and Procedures

Form	Syntax
`define`	`(define var expr)` or `(define (name params) body ...)`
`define-values`	`(define-values (var ...) expr)`
`lambda`	`(lambda (params) body ...)`
`case-lambda`	`(case-lambda (formals body ...) ...)`
`set!`	`(set! var expr)`

Conditionals and Sequencing

Form	Syntax
`if`	`(if test consequent alternate)`
`cond`	`(cond (test expr ...) ... (else expr ...))`
`case`	`(case key ((datum ...) expr ...) ... (else expr ...))`
`and`	`(and test ...)`
`or`	`(or test ...)`
`when`	`(when test body ...)`
`unless`	`(unless test body ...)`
`begin`	`(begin expr ...)`
`do`	`(do ((var init step) ...) (test result) body ...)`

Binding Forms

Form	Syntax
`let`	`(let ((var expr) ...) body ...)`
`let*`	`(let* ((var expr) ...) body ...)`
`letrec`	`(letrec ((var expr) ...) body ...)`
`letrec*`	`(letrec* ((var expr) ...) body ...)`
`let-values`	`(let-values (((v ...) expr) ...) body ...)`
`let*-values`	`(let*-values (((v ...) expr) ...) body ...)`

Quoting

Form	Syntax
`quote`	`'datum` or `(quote datum)`
`quasiquote`	`` `template `` with `,expr` and `,@expr`

Macros

Form	Syntax
`define-syntax`	`(define-syntax name (syntax-rules ...))`
`syntax-rules`	`(syntax-rules (literals) (pattern template) ...)`
`let-syntax`	`(let-syntax ((name transformer) ...) body ...)`
`letrec-syntax`	`(letrec-syntax ((name transformer) ...) body ...)`
`syntax-error`	`(syntax-error message args ...)`

Libraries and Program Structure

Form	Syntax
`define-library`	`(define-library (name ...) declarations ...)`
`import`	`(import import-set ...)`
`include`	`(include "file" ...)`
`include-ci`	`(include-ci "file" ...)` (case-insensitive)
`cond-expand`	`(cond-expand (feature-req body ...) ...)`

Exceptions, Records, Laziness, Parameters

Form	Syntax
`guard`	`(guard (var (test expr ...) ...) body ...)`
`define-record-type`	`(define-record-type name ctor pred field ...)`
`delay`	`(delay expr)`
`delay-force`	`(delay-force expr)`
`parameterize`	`(parameterize ((param val) ...) body ...)`

A.3 Numeric Procedures

Numbers are covered in Chapter 4.

Procedure	Description	Library
`+ - * /`	Arithmetic (variadic)	base
`= < > ⩽ ⩾`	Comparison (variadic, chained)	base
`zero? positive? negative?`	Sign predicates	base
`odd? even?`	Parity predicates	base
`abs`	Absolute value	base
`max min`	Maximum/minimum (variadic)	base
`quotient remainder modulo`	Integer division	base
`floor/ truncate/`	Division returning two values; single-value variants `floor-quotient`, `floor-remainder`, `truncate-quotient`, `truncate-remainder`	base
`gcd lcm`	Greatest common divisor, least common multiple	base
`numerator denominator`	Parts of a rational	base
`rationalize`	Simplest rational within a tolerance	base
`floor ceiling truncate round`	Rounding	base
`expt square`	Exponentiation, squaring	base
`exact-integer-sqrt`	Integer square root with remainder	base
`sqrt exp log sin cos tan atan`	Floating-point functions	inexact
`finite? infinite? nan?`	Floating-point predicates	inexact
`exact inexact`	Convert between exact and inexact	base
`exact? inexact? exact-integer?`	Test exactness	base
`number? integer? rational?`	Type predicates	base
`real? complex?`	Numeric tower predicates	base
`number→string string→number`	Conversion	base

A.4 Pair and List Procedures

All in `(scheme base)`; the compound accessors `caar` through `cddddr` are in `(scheme cxr)`. Lists are covered in Chapter 5.

Procedure	Description
`cons car cdr`	Pair construction and access
`set-car! set-cdr!`	Pair mutation
`list make-list`	Create a list
`length`	List length
`append`	Concatenate lists
`reverse`	Reverse a list
`list-ref list-tail list-set!`	Positional access and mutation
`map for-each`	Apply procedure to list elements
`member memv memq`	Search by `equal?/eqv?/eq?`
`assoc assv assq`	Association list lookup
`list-copy`	Shallow copy
`pair? null? list?`	Predicates

A.5 String Procedures

Strings are covered in Chapter 4.

Procedure	Description	Library
`string make-string`	Creation (from characters / of a given length)	base
`string-length`	Number of characters	base
`string-ref string-set!`	Character access and mutation	base
`substring`	Extract a portion	base
`string-append`	Concatenate strings	base
`string→list list→string`	Convert to/from character lists	base
`string-copy string-copy!`	Copy a string (into another)	base
`string-fill!`	Overwrite with a character	base
`string-map string-for-each`	Higher-order operations	base
`string=? string<? string>?`	Comparison (also `string`⩽?, `string`⩾?)	base
`string-ci=? string-ci<?`	Case-insensitive comparison (also >?, ⩽?, ⩾?)	char
`string-upcase string-downcase`	Case conversion	char
`string-foldcase`	Case folding for comparison	char
`string→symbol symbol→string`	Symbol conversion	base
`string→vector vector→string`	Vector conversion	base

A.6 Character Procedures

Procedure	Description	Library
`char-alphabetic? char-numeric?`	Classification	char
`char-whitespace?`	Whitespace test	char
`char-upper-case? char-lower-case?`	Case test	char
`char-upcase char-downcase`	Case conversion	char
`char-foldcase`	Case folding	char
`digit-value`	Numeric value of a digit	char
`char→integer integer→char`	Numeric conversion	base
`char=? char<? char>?`	Comparison (also `char` ⩽ ?, `char` ⩾ ?)	base
`char-ci=? char-ci<?`	Case-insensitive comparison (also >?, ⩽ ?, ⩾ ?)	char

A.7 Vector Procedures

All in `(scheme base)`. Vectors are covered in Chapter 4.

Procedure	Description
`vector make-vector`	Creation
`vector-ref vector-set!`	Access and mutation
`vector-length`	Size
`vector→list list→vector`	Conversion
`vector-copy vector-copy!`	Copy (into another vector)
`vector-append`	Concatenate vectors
`vector-fill!`	Overwrite with a value
`vector-map vector-for-each`	Higher-order operations
`vector?`	Type predicate

A.8 Bytevector Procedures

All in `(scheme base)`. Bytevectors hold raw bytes (integers 0–255) for binary data.

Procedure	Description
`bytevector make-bytevector`	Creation
`bytevector-length`	Size in bytes
`bytevector-u8-ref bytevector-u8-set!`	Access and mutation
`bytevector-copy bytevector-copy!`	Copy (into another bytevector)
`bytevector-append`	Concatenate bytevectors
`utf8→string string→utf8`	Convert to/from UTF-8 text
`bytevector?`	Type predicate

A.9 I/O Procedures

I/O is covered in Chapter 10. First, reading and writing data:

Procedure	Description	Library
`display write`	Output (human-readable / re-readable)	write
`write-shared write-simple`	Output with/without shared-structure labels (`#0=/#0#`, Chapter 5)	write
`read`	Read one datum	read
`newline write-char write-string`	Character/string output	base
`read-line read-char read-string`	Textual input	base
`peek-char char-ready?`	Lookahead	base
`read-u8 peek-u8 write-u8`	Binary I/O (byte at a time; `u8-ready?` tests availability)	base
`read-bytevector read-bytevector!`	Binary block input	base
`write-bytevector`	Binary block output	base

And the ports these operations read from and write to:

Procedure	Description	Library
`open-input-string`	String ports	base
`open-output-string`		
`get-output-string`	Extract string port contents	base
`open-input-bytevector`	Bytevector ports	base
`open-output-bytevector`		
`get-output-bytevector`	Extract bytevector port contents	base
`open-input-file`	File ports	file
`open-output-file`		
`open-binary-input-file`	Binary file port (input)	file
`open-binary-output-file`	Binary file port (output)	file
`call-with-input-file`	File ports with automatic close	file
`call-with-output-file`		
`with-input-from-file`	Redirect default ports	file
`with-output-to-file`		
`file-exists? delete-file`	File operations	file
`call-with-port`	Safe port handling	base
`close-port`	Close ports	base
`close-input-port`		
`close-output-port`		
`flush-output-port`	Flush buffered output	base
`current-input-port`	Default ports	base
`current-output-port`		
`current-error-port`	Error output	base
`eof-object eof-object?`	End-of-file value and test	base
`port? input-port?`	Port predicates	base
`output-port?`		
`binary-port? textual-port?`	Port kind predicates	base
`input-port-open?`	Test whether a port is still open	base
`output-port-open?`		

A.10 Control Flow Procedures

All in `(scheme base)`. Continuations are covered in Chapter 15, exceptions in Chapter 14.

Procedure	Description
`apply`	Call procedure with list of arguments
`values call-with-values`	Multiple return values
`call/cc`	Capture current continuation (`call-with-current-continuation`)
`dynamic-wind`	Entry/exit guarantees
`raise raise-continuable`	Signal exceptions
`error`	Create and raise an error object
`with-exception-handler`	Install exception handler
`error-object? error-object-message`	Inspect errors
`error-object-irritants`	Error context values
`read-error? file-error?`	Classify error objects

A.11 Boolean and Equivalence

All in `(scheme base)`. See Section 4.9 for choosing between the equality predicates.

Procedure	Description
`not`	Boolean negation
`boolean? boolean=?`	Type predicate, boolean comparison
`symbol=?`	Symbol comparison
`eq?`	Pointer identity
`eqv?`	Value equivalence (numbers, chars)
`equal?`	Deep structural equality

A.12 Type Predicates

All in `(scheme base)`.

Procedure	Description
`number? string? char? symbol?`	Atomic values
`pair? null? list? vector? bytevector?`	Compound values
`boolean? procedure? port?`	Other values
`eof-object? promise?`	Special values

A.13 Other Procedures

Procedure	Description	Library
`make-parameter` `parameterize`	Dynamic binding	base
`force make-promise` `promise?`	Lazy evaluation	lazy
`features`	List of feature identifiers	base
`eval environment`	Evaluation	eval
`interaction-environment`	The REPL environment	repl
`scheme-report-environment`	R5RS environments (also `null-environment`)	r5rs
`load`	Load a file	load
`exit emergency-exit`	Terminate the program	process-context
`command-line`	Program arguments	process-context
`get-environment-variable`	Read one env var	process-context
`get-environment-variables`	All env vars as an alist	process-context
`current-second` `current-jiffy`	Time	time
`jiffies-per-second`	Time resolution	time
`real-part imag-part` `magnitude angle`	Complex number accessors	complex

Appendix B
Kaappi CLI Reference

B.1 Synopsis

```
kaappi [OPTIONS] [FILE] [ARGS ... ]
kaappi compile FILE [-o OUTPUT]
kaappi check FILE
kaappi ast|expand|ir FILE
kaappi test [PATHS ... ]
kaappi fmt [--check] [FILES ... ]
kaappi explain CODE
kaappi doctor [--json]
kaappi features [--json]
kaappi cache status|clear
```

With no arguments, launches the REPL. With a file argument, executes the Scheme source file. Input can also be piped: `echo '(+ 1 2)' | kaappi`.

B.2 Commands

Command	Description
`kaappi FILE`	Run a Scheme source file
`kaappi`	Launch the interactive REPL
`kaappi compile FILE`	Compile Scheme source to a native binary via LLVM

The development-tool subcommands (`check`, `test`, `fmt`, `explain`, `doctor`, `features`, `ast`, `expand`, `ir`, `cache`) are covered in Section B.4.

B.3 Options

Flag	Description
`-h`, `–help`	Show usage and available flags
`–version`	Print version string
`–lib-path DIR`	Add a library search path (repeatable, up to 16)
`–compile FILE`	Compile to bytecode (`.sbc`) without running
`-o FILE`	Output path for `–compile`, `–emit-llvm`, or `compile`
`–emit-llvm`	Emit LLVM IR text (`.ll` file)
`–profile`	Profile execution (per-function timing, call counts)
`–profile-json FILE`	Write profiling data as JSON
`–sandbox`	Sandbox mode; see Section B.12
`–timeout MS`	Execution timeout in milliseconds
`–max-memory BYTES`	Maximum heap memory
`–disassemble FILE`	Show compiled bytecode without running
`–gc-stats`	Print GC statistics on exit
`–coverage`	Report library procedure coverage
`–coverage-xml FILE`	Write Cobertura XML coverage report
`–completions SHELL`	Output shell completion script (bash, zsh, or fish)
`–diagnostics=FMT`	Diagnostic output format: `text` (default) or `json`
`–timings[=FMT]`	Per-stage pipeline timings and cache HIT/MISS on stderr; `text` or `json`
`–deny-warnings`	With `check`: treat lint warnings as errors for the exit code
`–no-ir-opt`	Disable IR optimization passes (skips the bytecode cache)

With `–diagnostics=json`, every read, expand, compile, and runtime diagnostic is emitted as one JSON object per line on stderr, shaped like an LSP `Diagnostic`—the same objects the language server sends, so editor tooling and scripts can share one parser.

B.4 Development Tools

`kaappi check`: Static Analysis

Reads, expands, and compiles a file without executing it, reporting read and compile errors plus `KP4xxx` lint warnings: unknown top-level variables, and wrong arity or a wrong-typed literal in a direct call to a built-in. It never rejects a program R7RS permits, so a clean `check` is a cheap gate before running or committing:

```
$ kaappi check lint.scm
3:1: warning[KP4001]: unknown variable 'undefined-thing' at top level
check: 0 errors, 1 warning
```

Errors make the exit status non-zero; warnings do so only with `–deny-warnings`. `–diagnostics=json` applies here too.

`kaappi test`: Suite Runner

Runs SRFI-64 test files (Chapter 16)—each in its own subprocess, so one crash cannot take down the rest—and aggregates the runners' own tallies:

```
$ kaappi test tests/
kaappi test: seed 352045 (reproduce with: kaappi test --seed 352045)
  PASS  tests/test-math.scm  (2 tests, 0 skipped, 6ms)

Summary: 2 passed, 0 failed, 0 unexpected-pass, 0 expected-fail, 0 skipped
Files:   1 run, 0 failed, 0 errored (66ms)
```

`--json` emits the results as JSON, `--seed N` reproduces an ordering, and `--changed` (or `--list-affected`, optionally `--since REV`) runs only the suites whose import closure actually changed.

`kaappi fmt`: Formatter

Canonically formats Scheme source in place: two-space R7RS indentation and 80-column reflow, preserving comments. Before writing, it re-reads the result and checks it is `equal?` to the original program, so formatting can never change what the code means. `--check` reports files that need formatting and exits 1 without writing (for CI); with no files it formats stdin to stdout.

`kaappi explain`, `doctor`, and `features`

`explain CODE` prints a diagnostic code's registry entry—what it means, a minimal example that triggers it, and how to fix it (Chapter 14); `--all` lists every code. `doctor` runs the installation self-check described in Chapter 2. `features` reports the build's capabilities: version and build id, target triple, compiled-in subsystems, built-in versus portable SRFIs, and VM/GC limits—derived from the same source as `cond-expand` and `(features)`, so it cannot drift. All three take `--json`.

Pipeline Inspection: `ast`, `expand`, `ir`

Read-only dumps of successive compiler stages: `ast` prints the datums as read, `expand` prints the program after full macro expansion (valid source—it round-trips; Chapter 11), and `ir` prints the intermediate representation after the optimization passes (`--no-opt` shows it before).

B.5 Environment Variables

Variable	Description
`KAAPPI_HOME`	Overrides the `~/.kaappi` directory (libraries, packages, history, `config`, bytecode cache)
`KAAPPI_LIB_DIR`	Directory containing `libkaappi_rt.a` (used by `kaappi compile`)
`NO_COLOR`	When set and non-empty, disables all REPL coloring
`HOME`	Locates `~/.kaappi` when `KAAPPI_HOME` is unset

B.6 Library Search Order

When resolving `(import (lib name))`, Kaappi searches for `lib/name.sld` in:

1. Current working directory (`./`)
2. The `./lib/` subdirectory
3. Directories from `-lib-path` flags (in order)
4. The script's own directory, when running a file
5. `$KAAPPI_HOME/lib` (default `~/.kaappi/lib`, where thottam installs packages)
6. `../lib` relative to the `kaappi` executable (last-resort fallback for from-source builds)

B.7 The REPL

The REPL prompt is `kaappi>`. Expressions with unbalanced parentheses automatically continue on the next line. History is saved to `~/.kaappi/history` (up to 1000 entries). Exit with `,quit`, `(exit)`, or Ctrl-D on an empty line.

The REPL reads `~/.kaappi/config` at startup—a plain `key: value` file. The `repl.theme` key picks the `dark` (default) or `light` color preset; `repl.color.token` overrides one token class (`keyword`, `string`, `number`, `comment`, `boolean`, `paren`, `match-paren`, `prompt`, `continuation`) with a named color such as `cyan`, `bright-red`, `bold green`, or `none`; `repl.prompt` replaces the prompt text; `repl.history-length` resizes the history; and `repl.highlight: false` disables highlighting entirely (Chapter 2 shows an example file).

Comma-Commands

The REPL supports meta-commands prefixed with `,` (comma). The variable `_` holds the last result.

Command	Description
`,help`	Show all available commands
`,quit`	Exit the REPL
`,time expr`	Measure execution time
`,type expr`	Show the type of the result
`,describe sym`	Show procedure arity and type
`,apropos str`	Search bindings by substring
`,env [prefix]`	List bindings (optionally filtered by prefix)
`,expand expr`	Show macro expansion
`,dis expr`	Disassemble a procedure
`,profile expr`	Profile timing, calls, and allocations
`,break name`	Set a breakpoint on a function
`,condition id expr`	Set a conditional breakpoint
`,breakpoints`	List active breakpoints
`,delete all`	Clear all breakpoints
`,step expr`	Evaluate with single-stepping
`,gc`	Show GC statistics
`,version`	Show Kaappi version
`,load file`	Load and run a Scheme file
`,import lib`	Import a library (e.g., `,import (srfi 1)`)

B.8 Interactive Debugger

When a breakpoint fires, Kaappi enters an interactive `debug>` prompt with its own command set:

Command	Short	Description
`step`	`s`	Step into next expression
`next`	`n`	Step over
`finish`	`out`	Step out of current procedure
`continue`	`c`	Continue to next breakpoint
`locals`	`l`	Show local variables
`backtrace`	`bt`	Show call stack
`up`		Inspect caller frame
`down`		Inspect callee frame
`watch var`		Break on variable change
`unwatch var`		Remove a watch
`quit`	`q`	Exit debugger

B.9 Thottam Package Manager

```
thottam COMMAND [ARGS]
```

Command	Description
`thottam install PKG`	Install a package and its dependencies
`thottam remove PKG`	Remove an installed package
`thottam list`	List all installed packages
`thottam update`	Update all packages to latest versions
`thottam update PKG`	Update a specific package
`thottam verify`	Check installed packages against the lockfile
`thottam –locked install PKG`	Refuse to install packages not in the lockfile
`thottam –version`	Show thottam version
`thottam –completions SHELL`	Output shell completions (bash, zsh, fish)

`install` accepts an optional version pin and source URL:

```
thottam install PKG[@VER][::URL]

thottam install kaappi-web              # latest release
thottam install kaappi-web@v1.0.0       # exact tag
thottam install kaappi-net@"≥0.2.0"     # version range
thottam install kaappi-auth::https://github.com/bob/kaappi-auth
```

Packages are installed to `~/.kaappi/lib/`, and the lockfile is kept at `~/.kaappi/lock.json`.

B.10 Native Compilation

Compile a Scheme program to a standalone native binary:

```
kaappi compile program.scm -o myapp
./myapp
```

Requirements:

- LLVM must be available on the system (the compiler emits LLVM IR and invokes `clang`)
- `libkaappi_rt.a` must be built (`zig build lib`)

To inspect the generated LLVM IR without compiling:

```
kaappi --emit-llvm program.scm -o program.ll
```

B.11 Bytecode Caching

When Kaappi runs a file, it caches the compiled bytecode (`.sbc`) under `$KAAPPI_HOME/cache` (default `~/.kaappi/cache`). On subsequent runs of an unchanged file, the cached bytecode is loaded directly—skipping the reader, expander, and compiler passes. The cache key includes the exact compiler build, so upgrading Kaappi (or rebuilding it from source) recompiles rather than serving bytecode an older compiler produced.

```
kaappi cache status   # location, entries, sizes
kaappi cache clear    # remove all cached bytecode
```

Add `–timings` to any run to see the per-stage cost and an explicit cache `HIT` or `MISS` line.

B.12 Sandbox Mode

`–sandbox` restricts what a program can do:

- FFI is disabled (`ffi-open`, `ffi-fn`, and the other `ffi-*` procedures raise errors)
- File I/O is disabled (`open-input-file`, `call-with-output-file`, `file-exists?`, `delete-file`, and the rest of `(scheme file)`)
- All SRFI-170 filesystem operations are disabled
- `eval` and `load` are disabled
- Process context is blocked (`command-line`, `exit`, environment variables)
- OS threads (SRFI-18) are disabled

Pure computation, standard output (`display`, `write`), and green fibers remain available.

Useful for running untrusted code safely:

```
kaappi --sandbox --timeout 5000 --max-memory 50000000 untrusted.scm
```

Appendix C
SRFI Catalog

All 72 SRFIs supported by Kaappi, grouped by category. Import with `(import (srfi N))`. Chapter 16 covers the most commonly used SRFIs in depth. Each SRFI's full specification, rationale, and reference implementation are published at `https://srfi.schemers.org`.

Built-in vs. Portable. Of Kaappi's 72 SRFIs, 8 are compiled into the interpreter binary (marked with * below) for performance. The remaining 64 are portable Scheme `.sld` files loaded on demand. Both are imported the same way: `(import (srfi N))`.

C.1 Data Structures

SRFI	Description
1*	Extended list library (fold, filter, find, partition, iota, zip, take, drop)
4	Homogeneous numeric vector datatypes
9*	Defining record types (define-record-type)
69*	Basic hash tables (make, set!, ref, delete!, walk, keys, values)
111	Boxes (single-value mutable containers)
113	Sets and bags with comparator-based equality
116	Immutable list library (functional pairs)
117	Queues based on lists
125	Intermediate hash tables (richer API than SRFI-69)
128	Comparators (for use with sorted/hashed containers)
132	Sort libraries (list-sort, vector-sort, merge)
133*	Vector library (vector-map, vector-fold, vector-filter, vector-any)
134	Immutable deques (double-ended queues)
146	Mappings (functional key-value maps, also hash variant)
195	Multiple-value boxes
196	Range objects (lazy integer ranges)
222	Compound objects

C.2 Strings and Text

SRFI	Description
6	Basic string ports (open-input-string, open-output-string)
13*	String library (contains, split, join, trim, prefix?, suffix?)
14	Character sets (char-set, char-set-contains?)
28	Basic format strings (~a, ~s placeholders)
48	Intermediate format strings (numbers, padding)
115	Scheme regular expressions (S-expression syntax)
130	Cursor-based string library
152	String library (reduced, R7RS-compatible subset)
166	Formatting (pretty-print, columnar, unicode, color)
175	ASCII character library
233	INI file parser

C.3 Numbers and Bits

SRFI	Description
27	Sources of random bits (random-integer, random-real)
60	Integers as bits (bitwise-and, bitwise-or, bit-count)
141	Integer division (floor/, ceiling/, truncate/, round/)
143	Fixnums (fixnum-specific arithmetic)
144	Flonums (flonum-specific arithmetic)
151	Bitwise operations (comprehensive bit manipulation)

C.4 Control Flow and Syntax

SRFI	Description
0	Feature-based conditional expansion (cond-expand)
2	`and-let*` — short-circuit `let` with guards
8	`receive` — bind formals from multiple values
11	`let-values` and `let*-values`
16	`case-lambda` — procedures with variable arity
17	Generalized `set!`
26	`cut` and `cute` — partial application notation
31	`rec` — recursive evaluation shorthand
39*	Parameter objects (dynamic binding)
61	A more general `cond` clause
87	⇒ in `case` clauses
145	`assume` — assertions with optimizer hints
197	Pipeline operator
210	Procedures and syntax for multiple values
219	Define higher-order lambda
227	Optional arguments
232	Flexible curried procedures
235	Combinators (compose, constantly, complement)

C.5 Iteration, Generators, and Lazy Evaluation

SRFI	Description
41	Streams (lazy lists with memoization)
42	Eager comprehensions (list-ec, vector-ec, sum-ec)
45	Primitives for expressing iterative lazy algorithms
127	Lazy sequences
158	Generators and accumulators (lazy pull-based iteration)

C.6 Exceptions and Conditions

SRFI	Description
23	Error reporting (`error` procedure)
34	Exception handling for programs
35	Conditions (structured exception types)
36	I/O conditions
189	`maybe` and `either` — optional values without exceptions

C.7 Time, System, and Concurrency

SRFI	Description
18*	Multithreading (threads, mutexes, condition variables)
19	Time data types and procedures (date, time-utc)
37	`args-fold` — program argument processor
98	Interface to access environment variables
170*	POSIX API (file-info, directories, symlinks, permissions)
174	POSIX timespecs

C.8 Testing

SRFI	Description
64	A Scheme API for test suites (test-begin, test-assert, test-equal)
78	Lightweight testing (check, check-report)

C.9 Miscellaneous

SRFI	Description
38	External representation for data with shared structure
43	Vector library (R7RS-compatible, superseded by 133)

Appendix D
Common Error Messages

This appendix lists the most frequent error messages you will encounter while learning Kaappi, along with their causes and fixes. Every message shown here is the actual output of the interpreter. Errors fall into three categories—read-time, compile-time, and runtime—plus library and FFI errors, and every diagnostic carries a stable `KP` code you can look up.

D.1 How Kaappi Reports Errors

When an error occurs while running a file, Kaappi prints the source location, a diagnostic code, the message, and the offending line:

```
program.scm:1:1: error[KP3001]: undefined variable 'dispaly'.
    Did you mean 'display'?
    (dispaly "hi")
```

Four things to know about this format:

- **Location prefix.** `program.scm:1:1:` names the file, line, and column of the offending expression—down to the exact sub-form—and the source line is echoed beneath the message. When the failure happens inside a procedure, a `called from` line points at the call site. In the REPL the location prefix is omitted.
- **Diagnostic codes.** The `[KP3001]` tag identifies the diagnostic independently of its wording: `KP1xxx` are read-time, `KP2xxx` compile-time and library, `KP3xxx` runtime, and `KP4xxx` the lint warnings of `kaappi check`. Run `kaappi explain KP3001` for any code's meaning, a minimal example, and the fix; programs can dispatch on codes with `error-object-code` (Section 14.2).

- **Did-you-mean suggestions.** For misspelled variables, Kaappi suggests the closest known binding. The suggestion depends on what is in scope, so this appendix quotes only the stable part of each message.

- **Machine-readable output.** `–diagnostics=json` emits every diagnostic as LSP-shaped JSON on stderr for editors and scripts (Appendix B).

A raised exception that reaches the top level is reported with the error object's message and irritants, e.g. `error[KP3004]: division by zero`. Handlers can also read both fields programmatically with **`guard`**:

```
> (guard (e (#t (display (error-object-message e))))
    (/ 1 0))
division by zero
```

D.2 Read-Time Errors

The reader reports these when it cannot parse your source into Scheme data, with a `file:line:col:` prefix.

`read error[KP1001]: unexpected end of input`

Cause: Input ended while a list, string, or block comment was still open. Usually a missing closing parenthesis.

```
(define (square x)
  (* x x)
; program.scm:3:1: read error[KP1001]:
;   unexpected end of input
```

Fix: Add the missing `)`. Use an editor with bracket matching; the REPL's multi-line mode tracks open parens for you.

`read error[KP1003]: unexpected ')'`

Cause: More closing `)` than opening `(`.

```
(display (+ 1 2)))
; program.scm:1:19: read error[KP1003]:
;   unexpected ')'
```

Fix: Remove the extra parenthesis.

`read error[KP1006]: unterminated string literal`

Cause: A string literal is missing its closing double quote.

Fix: Add the closing `"`, or check for unescaped quotes inside the string. Use `\"` to include a literal quote.

`read error[KP1007]: invalid escape sequence`

Cause: A string contains an escape sequence Kaappi does not recognize, such as `\q`. Valid escapes are `\n`, `\r`, `\t`, `\a`, `\b`, `\\`, `\"`, `\|`, and `\x<hex>;`.

Fix: Use a supported escape. For hex escapes, write hex digits followed by a semicolon: `\x41;`.

D.3 Compile-Time Errors

`compile error[KP2001]: invalid syntax`

Cause: A special form is malformed—`let` with no bindings or body, `lambda` with no parameter list, `if` with no branches, `define` with no value.

```
> (lambda)
; compile error[KP2001]: invalid syntax
```

Fix: Supply all required sub-expressions. `lambda` needs at least a parameter list and a body: `(lambda (x) x)`.

D.4 Runtime Errors

`error[KP3001]: undefined variable 'foo'`

Cause: You referenced a name that has no binding. When a similarly spelled binding exists, Kaappi appends a suggestion: `Did you mean 'log'?`

Common reasons:

- Typo in the variable name (e.g., `lenght` instead of `length`)—follow the did-you-mean hint.
- Used a name before its `define` (order matters in scripts).
- The name comes from a *portable* SRFI or an ecosystem library you have not imported. Kaappi's eight built-in SRFIs (including SRFI-1's `filter` and

`fold`) are always available, but the 64 portable SRFIs load from `.sld` files only when imported—`list-sort` needs `(import (srfi 132))`. Appendix C marks which SRFIs are built in.

Fix: Check the spelling, reorder definitions, or add the missing `(import …)`.

The `set!` variant of this error reads `error[KP3001]: set!: unbound variable 'y'`—`set!` requires an existing binding, so use **`define`** for the initial one.

error[KP3002]: type error in 'car': expected pair, got ()

Cause: A primitive received an argument of the wrong type. The message names the procedure, the expected type, and the offending argument—here, `car` applied to the empty list. Some primitives, such as the arithmetic operators, show the argument's type rather than its value:

```
> (car '())
; error[KP3002]: type error in 'car':
;   expected pair, got ()
> (+ 1 "two")
; error[KP3002]: type error in 'arithmetic':
;   expected number, got #<string>
```

Fix: Check for the empty list with `null?` before calling `car`/`cdr`. For arithmetic on strings, convert first with `string→number`.

error[KP3003]: 'f': expected 2 arguments, got 1

Cause: A procedure was called with the wrong number of arguments. Variadic procedures report `expected at least N arguments`.

Fix: Check the procedure's signature. Use `,describe f` in the REPL to see its arity. `kaappi check` catches wrong-arity calls to built-ins before the program runs.

error[KP3005]: not a procedure

Cause: The first element of a call expression evaluated to something that is not callable: `(42 1)`, or `(x y)` where `x` holds a number.

Fix: Check for extra parentheses—`((+ 1 2))` computes 3 and then tries to call it.

`error[KP3004]: division by zero` (uncaught exceptions)

Cause: An exception raised by `error`, `raise`, or a primitive (such as division by zero) reached the top level uncaught. The report shows the error object's message followed by its irritants, if any. Exceptions the runtime raises carry their own specific code, like `KP3004` here; error objects your code creates with `error` are reported under the generic `KP3000`:

```
> (/ 1 0)
; error[KP3004]: division by zero
> (error "bad input" 42)
; error[KP3000]: bad input 42
```

Fix: Wrap the code in **`guard`** to catch the exception and handle it (see Chapter 14). To avoid dividing by zero in the first place, test the divisor: `(if (zero? b) ... (/ a b))`.

`error[KP3006]: vector-ref: index 5 out of range for length 1`

Cause: `vector-ref`, `string-ref`, or `bytevector-u8-ref` called with an index beyond the length. The message names the procedure, the out-of-range index, and the sequence length.

Fix: Check that the index is between 0 and the sequence length minus one. Use the matching length procedure to verify bounds.

`error[KP3008]: stack overflow`

Cause: Non-tail-recursive function with very deep recursion. The call stack grows automatically up to 32768 frames; only recursion deeper than that overflows.

Fix: Rewrite as a tail-recursive function using an accumulator or named `let`. See Chapter 7 for the tail-call pattern.

D.5 Library and Import Errors

`error[KP2001]: library not found: (mylib.utils)`

Cause: Kaappi cannot find the file `mylib/utils.sld` on any search path. (The message prints the library name with dots.)

Fix: Ensure the `.sld` file exists at the correct path relative to your working directory, `./lib/`, or a `--lib-path` directory. Ecosystem libraries are installed

with `thottam install` and auto-discovered from `~/.kaappi/lib/`. See Appendix B for the full search order.

`error[KP2001]: circular import: (mylib.a)`

Cause: Library A imports B, and B imports A.

Fix: Extract shared definitions into a third library C that both A and B import.

D.6 FFI Errors

`error[KP3002]: ffi-open: dlopen( ... )`

Cause: `ffi-open` cannot find or load the shared library file (`.dylib` on macOS, `.so` on Linux). The message passes along the system loader's own report—`dlopen` on macOS lists every path it tried.

Fix: Build the native library first (`make` in its directory), then use the full path, pass `–lib-path`, or set `DYLD_LIBRARY_PATH` (macOS) / `LD_LIBRARY_PATH` (Linux).

`error[KP3002]: ffi-fn: symbol not found`

Cause: The shared library loaded, but does not contain the requested function. The message includes the loader's own report (`dlsym` on macOS).

Fix: Check the function name for typos. List the library's exported symbols with `nm -gU lib.dylib` (macOS) or `nm -D lib.so` (Linux).

`error[KP2001]: sandbox: cannot load library from file`

Cause: Running with the `–sandbox` flag, which blocks loading libraries from files—including `(kaappi ffi)` itself, so sandboxed programs cannot reach the FFI at all (file I/O, `eval`, and more are also disabled; see Appendix B).

Fix: Remove `–sandbox` if you need FFI access.

D.7 Pitfalls That Are Not Errors

Behavior that silently surprises programmers coming from Python or JavaScript:

- **Only `#f` is false.** `0`, `""`, and `'()` are all true in conditionals.
- **`eq?` does not compare numbers reliably.** Use `=` for numbers, `equal?` for structures. See Section 4.9.

- **`set!` changes the binding, not the caller's variable.** Assigning to a parameter inside a procedure does not affect the argument at the call site.
- **String literals are immutable.** `string-set!` needs a mutable string; make one with `string-copy`.
- **`map` stops at the shortest list.** With lists of different lengths, extra elements are silently ignored.

D.8 REPL Tips for Debugging

Command	Purpose
`,type expr`	Show the type of a value
`,describe name`	Show procedure arity and type
`,expand expr`	Show macro expansion
`,dis expr`	Disassemble a procedure
`,break name`	Set a breakpoint on a function
`,step expr`	Evaluate with single-stepping

Breakpoints drop you into the interactive debugger; its command set is listed in Appendix B. Outside the REPL, `kaappi check file.scm` reports compile-time problems and lint findings without running the program, and `kaappi explain CODE` documents any diagnostic this appendix does not cover.

Appendix E
Ecosystem Packages

Kaappi's package manager `thottam` provides access to a growing ecosystem of libraries. Install any package with:

```
thottam install package-name
```

Dependencies are resolved automatically—installing `kaappi-web` also pulls in `kaappi-http` and `kaappi-json`. Packages land in `~/.kaappi/lib/` and are immediately importable from any Scheme file. All packages are MIT licensed. For the full set of `thottam` commands (removing, updating, pinning versions), see Appendix B.

Native Packages Need System Libraries. Packages that wrap C libraries via the FFI compile a small native shim when installed. OpenSSL is needed by `kaappi-net` and `kaappi-crypto`, libpq by `kaappi-pg`, and libsqlite3 by `kaappi-sqlite`. The pure-Scheme packages have no system requirements.

E.1 Networking and Web

Package	Description	Import
kaappi-net	TCP client/server, TLS client	`(kaappi net)`
kaappi-http	HTTP/HTTPS client and server	`(kaappi http)`
kaappi-web	Web framework (routing, middleware, JSON helpers)	`(kaappi web)`
kaappi-template	Text and HTML template engine with auto-escaping	`(kaappi template)`
kaappi-email	SMTP email sending	`(kaappi email)`

E.2 Data Formats

Package	Description	Import
kaappi-json	JSON reading and writing	`(kaappi json)`
kaappi-toml	TOML parser and emitter	`(kaappi toml)`
kaappi-yaml	YAML parser and emitter	`(kaappi yaml)`
kaappi-csv	RFC 4180 CSV parsing and generation	`(kaappi csv)`

E.3 Databases

Package	Description	Import
kaappi-pg	PostgreSQL client (queries, params, pools)	`(kaappi pg)`
kaappi-sqlite	SQLite3 embedded database	`(kaappi sqlite)`
kaappi-redis	Redis client (commands, pub/sub)	`(kaappi redis)`

E.4 Utilities and Testing

Package	Description	Import
kaappi-crypto	Hashing (SHA-256, SHA-512, MD5), HMAC	`(kaappi crypto)`
kaappi-log	Structured logging with levels	`(kaappi log)`
kaappi-cli	CLI argument parsing, subcommands, help	`(kaappi cli)`
kaappi-test	Unit testing (assertions, test suites, CI)	`(kaappi test)`
kaappi-bdd	BDD-style testing (describe, it, expect)	`(kaappi bdd)`
kaappi-examples	Example applications demonstrating real-world usage	(various)

E.5 Quick Start Examples

Reading JSON

JSON objects parse to association lists, arrays to lists—look up keys with `assoc`:

```
(import (kaappi json))

(define data
  (call-with-input-file "config.json" json-read))
(display (cdr (assoc "name" data)))
```

HTTP Request

```
(import (kaappi http))
```

```
(define response (http-get "https://api.example.com/data"))
(display (response-body response))
```

PostgreSQL Query

Each row comes back as a vector:

```
(import (kaappi pg))

(define conn (pg-connect "host=localhost dbname=mydb"))
(define rows (pg-query conn "SELECT name, age FROM users"))
(for-each (lambda (row)
            (display (vector-ref row 0))
            (newline))
          rows)
(pg-close conn)
```

CLI Tool

Flags and options take a short name, a long name, and a description; parsed values are read back by their long name:

```
(import (kaappi cli))

(define app
  (cli "greet" "A greeting tool"
    (flag "-l" "--loud" "Use uppercase")
    (option "-n" "--times" "Repeat N times" 1)
    (argument "name" "Name to greet")))

(define result (run-cli-parse app '("-l" "Ada")))
(parsed-flag? result "loud")                ; => #t
(cdr (assoc "name" (parsed-args result)))   ; => "Ada"
(parsed-ref result "times")                 ; => 1 (default)
```

Appendix F
Glossary

---

alist
: An *association list*: a list of pairs used as a key-value map. Keys are in the `car` of each pair, values in the `cdr`. Lookup is O(n) via `assoc` or `assq`.

arity
: The number of arguments a procedure accepts. In the REPL, `,describe` shows a procedure's arity.

atom
: A value that is not a pair: numbers, strings, symbols, characters, booleans, vectors, and bytevectors are all atoms.

bignum
: An arbitrary-precision integer. Kaappi automatically promotes fixnums to bignums on overflow.

binding
: An association between an identifier and a value, created by `define`, `let`, or a procedure call. Chapter 8 covers local bindings.

bytecode
: The low-level instruction format that Kaappi's compiler produces. Executed by the register-based virtual machine.

bytevector
: A fixed-length sequence of bytes (integers 0–255), created with `bytevector` or `make-bytevector`. Used for binary data and I/O buffers.

call/cc
: Short for `call-with-current-continuation`. Captures the current continuation as a first-class procedure that can be invoked later (Chapter 15).

car The first element of a pair. From "Contents of the Address Register" on the IBM 704.

cdr The second element of a pair. From "Contents of the Decrement Register." For a proper list, the cdr is the rest of the list.

channel
A communication primitive for fibers, created with `make-channel`. A channel is an unbounded queue: `channel-send` never blocks, while `channel-receive` waits until a value is available (Chapter 18).

closure
A function bundled with the environment in which it was defined. Closures capture free variables from their enclosing scope (Chapter 7).

cond-expand
A form for conditional compilation. Selects code branches based on features available in the implementation.

condition
A value raised as an exception. Can be an error object (created by `error`) or any Scheme value (raised by `raise`).

cons cell
A pair: a heap-allocated object with a `car` and a `cdr`. The fundamental building block of lists.

continuation
The "rest of the computation" from a given point in the program. Scheme allows capturing continuations as first-class values (Chapter 15).

datum
Any Scheme value that can be read by the reader: a number, string, symbol, list, vector, etc. A datum is the external representation of a value.

define-library
The R7RS form for creating a library (module). Specifies exports, imports, and the library body (Chapter 12).

dotted pair
A pair whose cdr is not a list. Printed as `(a . b)`.

dynamic extent
The period during which control is inside a given expression's evaluation—from entry until exit, including re-entries via continuations. `dynamic-wind` and `parameterize` operate on dynamic extents.

dynamic-wind
: A mechanism that guarantees setup and cleanup thunks are called when control enters or leaves a dynamic extent, even via continuations.

environment
: The set of bindings visible at a point in a program. Closures capture the environment where they were created.

eof object
: A special value returned by input procedures when end-of-file is reached. Tested with `eof-object?`.

exact
: A number whose value is represented precisely (integers, rationals). Contrasted with inexact (floating-point).

expander
: The compiler phase that rewrites macro uses into core forms before compilation. Scheme's expander is *hygienic*: expansion cannot accidentally capture user variables.

export
: A declaration in a library that makes a binding available to importers. Specified with the `export` clause in `define-library` (Chapter 12).

FFI (Foreign Function Interface)
: A mechanism for calling C functions from Scheme. Kaappi provides `(kaappi ffi)` with `ffi-open`, `ffi-fn`, and `ffi-callback` (Chapter 17).

fiber
: A lightweight green thread managed by Kaappi's runtime. Created with `spawn` from `(kaappi fibers)`. Fibers are cooperatively scheduled (Chapter 18).

fixnum
: A small integer stored inline (no heap allocation) in Kaappi's NaN-boxed value representation. 48-bit signed range (up to $\pm 2^{47}$).

flonum
: An IEEE 754 double-precision floating-point number. Inexact by definition.

form
: A Scheme expression, particularly one with special syntax (e.g., `if`, `define`, `let`).

green thread
: A thread scheduled by the language runtime rather than the operating system. See *fiber*.

guard clause
: An exception-handling form that catches raised exceptions and dispatches on conditions, similar to `try/except` in Python (Chapter 14).

hash table
: A mutable key-value data structure with O(1) average lookup. Provided by SRFI-69. Created with `make-hash-table`.

hygienic macro
: A macro system where generated identifiers cannot accidentally capture variables from the macro's call site. Scheme's `syntax-rules` is hygienic (Chapter 11).

identifier
: A name in Scheme source code. Can contain letters, digits, and extended characters like `-`, `?`, and `!`.

improper list
: A chain of pairs not terminated by the empty list. The last cdr is a non-list value.

inexact
: A number whose value may be approximate (floating-point). Created by decimal literals or operations that introduce rounding.

irritants
: Additional values attached to an error object, providing context about what went wrong.

lexical scope
: A scoping rule where a name refers to the binding in the nearest enclosing scope where it was defined. Also called static scope.

macro
: A syntax transformation that rewrites code before it is compiled. Defined with `define-syntax` and `syntax-rules` (Chapter 11). See also *hygienic macro*.

NaN-boxing
: Kaappi's value representation technique: all Scheme values are stored in 64-bit words using the unused bit patterns of IEEE 754 NaN values.

pair
: See *cons cell*.

parameter object
: A dynamically scoped variable created with `make-parameter` and rebound for the extent of a body with `parameterize`. Calling the parameter as a procedure returns its current value.

port
: An abstraction for I/O sources and sinks. Input ports read data; output ports write data. Ports can be connected to files, strings, or the console (Chapter 10).

predicate
: A procedure that returns `#t` or `#f`. By convention, predicates end with `?` (e.g., `null?`, `string?`).

procedure
: A first-class callable value. Created by `lambda`, `define`, or built-in.

promise
: A deferred computation created by `delay`. Its value is computed at most once, when `force` is called.

proper list
: A chain of pairs terminated by the empty list `'()`. The standard list type in Scheme.

quasiquote
: Template syntax using backtick (`` ` ``) that allows selective unquoting with `,` and splicing with `,@`.

R7RS
: The Revised^7 Report on the Algorithmic Language Scheme (2013). The current Scheme standard. "R7RS-small" is the core language; "R7RS-large" is the extended library effort.

record
: A user-defined structured data type with named fields. Created with `define-record-type` (Chapter 13).

REPL
: Read-eval-print loop: the interactive `kaappi>` prompt that reads an expression, evaluates it, and prints the result. Supports meta-commands prefixed with a comma (Appendix B).

S-expression
: The syntactic representation of Scheme code and data: atoms and nested lists in parentheses.

sandbox
: A restricted execution mode (`-sandbox`) for untrusted code. Disables the FFI, file I/O, `eval`, `load`, process context access, and OS threads (Appendix B).

shared library
: A compiled C library (`.dylib` on macOS, `.so` on Linux) that can be loaded at runtime via the FFI.

spawn
: The procedure that creates a new fiber from a thunk. The fiber runs concurrently with other fibers on the same OS thread.

special form
: An expression evaluated by its own rules rather than as a procedure call—`if`, `define`, and `lambda` are special forms. Appendix A lists them all.

SRFI
: Scheme Requests for Implementation. Community-designed portable libraries. Kaappi ships 72 SRFIs (8 built-in, 64 portable).

symbol
: An interned identifier value. Two symbols with the same name are pointer-equal (`eq?`). Used as keys, tags, and in code representation.

syntax-rules
: The pattern-based macro definition form in R7RS. Matches input against patterns and produces output from templates.

tail call
: A procedure call in *tail position*—the last expression evaluated before returning. Scheme guarantees tail calls run in constant stack space, a property known as tail-call optimization (Chapter 7).

TCO
: Tail Call Optimization. See *tail call*.

thottam
: Kaappi's built-in package manager. Installs, manages, and resolves dependencies for Kaappi ecosystem libraries.

thunk
: A zero-argument procedure, used to delay evaluation. `(lambda () expr)`.

truthiness
: In Scheme, only `#f` is false; every other value—including `0`, `""`, and the empty list—is true. A frequent trap for programmers coming from Python or JavaScript.

variadic
: A function that accepts a variable number of arguments. Defined with dotted parameter syntax: `(define (f x . rest) ...)`.

vector
: A fixed-size, mutable, indexed collection with O(1) access by position.

yield
: Voluntarily suspends the current fiber so other fibers can run. Implicitly called by `channel-receive` while it waits for a value.

Appendix G
Further Reading

This appendix collects books, standards documents, and online resources for continuing your study of Scheme and Kaappi.

G.1 Books

Structure and Interpretation of Computer Programs *Harold Abelson, Gerald Jay Sussman, and Julie Sussman* (MIT Press, 1996). The classic introduction to computer science using Scheme. Covers abstraction, recursion, interpreters, and compilation. Available free online from MIT Press.

The Scheme Programming Language, 4th Edition *R. Kent Dybvig* (MIT Press, 2009). A thorough reference and tutorial for the Scheme language. Covers R6RS but most content applies to R7RS. Available free online at `https://scheme.com/tspl4/`.

The Little Schemer *Daniel P. Friedman and Matthias Felleisen* (MIT Press, 1996). A short, playful introduction to recursive thinking using Scheme. Structured as a series of questions and answers that build up to writing a Scheme interpreter in Scheme. Highly recommended if you want to deepen your intuition for recursion.

The Seasoned Schemer *Daniel P. Friedman and Matthias Felleisen* (MIT Press, 1996). The sequel to *The Little Schemer*. Covers continuations, mutation, and advanced control flow.

How to Design Programs, 2nd Edition *Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi* (MIT Press, 2018). A systematic introduction to program design using a Scheme-family teaching language. Gentler than SICP. Available free online at `https://htdp.org`.

Essentials of Programming Languages, 3rd Edition *Daniel P. Friedman and Mitchell Wand* (MIT Press, 2008). Uses Scheme to build interpreters for progressively more complex languages. Excellent for understanding how programming languages work.

G.2 Standards and Specifications

R7RS-small Report The official language standard that Kaappi implements. Defines all syntax forms, standard libraries, and semantics. Available at `https://r7rs.org`. Essential reading once you are comfortable with the language. A follow-on effort, R7RS-large, is standardizing a much larger library set on top of it.

SRFI Process Scheme Requests for Implementation—the community process for proposing and standardizing library extensions. Browse all SRFIs at `https://srfi.schemers.org`. Each SRFI includes a rationale, specification, and reference implementation. Appendix C catalogs the 72 SRFIs Kaappi ships.

G.3 Kaappi Resources

Documentation Site `https://kaappi-lang.org` — the complete Kaappi reference, including the guided tutorial, language quick reference, cookbook, and procedure documentation organized by category (600+ procedures).

Tutorial `https://kaappi-lang.org/guide/tutorial/` — a 12-section beginner tutorial that complements this book.

Tour `https://kaappi-lang.org/tour/` — 12 interactive lessons that run live code in your browser. A quick way to refresh a concept without leaving the page.

Cookbook `https://kaappi-lang.org/cookbook/` — practical recipes for common tasks: building REST APIs, parsing JSON, working with databases, writing CLI tools, and testing.

Playground `https://kaappi-lang.org/playground/` — a browser-based REPL running Kaappi via WebAssembly. Useful for quick experiments without a local install.

Community `https://kaappi-lang.org/community/` — where to ask questions, report bugs, and follow development.

Source Code `https://github.com/kaappi/kaappi` — the Kaappi implementation in Zig. Reading the source is the best way to understand how the language works at a deep level.

G.4 Online Resources

Scheme Community Wiki `https://community.schemewiki.org` — tutorials, implementation comparisons, and community-maintained documentation.

Awesome Scheme `https://github.com/schemedoc/awesome-scheme` — a curated list of Scheme implementations, libraries, and resources.

Scheme and Lisp Communities `https://reddit.com/r/scheme` and `https://reddit.com/r/lisp` — active forums for discussing Scheme and Lisp topics.

Index