MASTERING RACKET

FROM FIRST STEPS TO LANGUAGE-ORIENTED PROGRAMMING

- CORE LANGUAGE
- FUNCTIONAL & OBJECT-ORIENTED PARADIGMS
- MACROS & METAPROGRAMMING
- CONTRACTS & RELIABILITY
- CONCURRENCY & PARALLELISM
- WEB DEVELOPMENT
- LANGUAGE DESIGN
- BUILD CUSTOM LANGUAGES

PRACTICAL EXAMPLES. REAL-WORLD PATTERNS.
UNDERSTAND HOW, WHY, AND WHEN.

JACOB LEE

Mastering Racket

From First Steps to Language-Oriented Programming

Steve Publications

This book is available at https://leanpub.com/masteringracket

This version was published on 2026-08-18

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

From First Steps to Language-Oriented Programming

This book takes you from your first Racket expression through advanced metaprogramming and language design. You will learn the core language, functional and object-oriented paradigms, macros, contracts, concurrency, web development, and how to create custom languages on top of Racket. Every chapter builds on the last with complete, runnable examples that demonstrate real-world usage patterns. By the end you will understand not only how to program in Racket but why it is designed the way it is and when its unique strengths make it the right tool for the job.

Introduction: Why Racket?

The Racket Promise: One Language to Rule Them All

Imagine a language where you can write a quick script in seconds, build a web server in minutes, create a graphical desktop application in an afternoon, and then extend or replace the language itself when none of those approaches quite fit your problem. That is not science fiction and it is not marketing hyperbole. That is Racket.

Racket is a general-purpose programming language with a distinctive philosophy: you should be able to make the language work for you, not force yourself to work within the language's fixed constraints. This idea, called language-oriented programming, means that when you encounter a problem domain that does not fit neatly into standard control structures or data types, you do not reach for yet another framework or embed SQL in your code. Instead, you extend Racket itself with new syntax, new semantics, and new abstractions tailored precisely to your needs.

This book will show you how. We start at the very beginning with installation, basic syntax, and simple programs. We progress through functions, data structures, modules, and contracts until you are comfortable writing idiomatic Racket code. Then we dive into what makes Racket truly special: macros that let you invent new syntactic forms, the #lang system that lets you define entirely new languages, and powerful tools for building domain-specific solutions. Along the way we cover practical topics like web development, databases, concurrency, GUI applications, testing, performance optimization, and deployment.

Where Racket Shines and Where It Does Not

Racket is not a universal solution and being honest about its limitations is as important as understanding its strengths.

Racket excels at:

- **Language creation and DSLs**: Racket's macro system and #lang protocol make it arguably the most capable mainstream language for creating custom syntax and domain-specific languages. Game studios use Racket-based languages to author content for AAA titles, researchers prototype new programming languages on top of it, and engineers build internal tools with bespoke syntax that matches their domain vocabulary.
- **Rapid prototyping**: The interactive REPL, excellent standard library, and concise syntax let you explore ideas quickly. You can go from concept to working prototype in a fraction of the time required by more verbose languages.
- **Education and teaching**: Racket was designed with pedagogy in mind from its inception. Its gradual language levels help students learn programming concepts incrementally without being overwhelmed. Many universities use Racket for introductory computer science courses.
- **Research and experimentation**: Language researchers love Racket because it lets them implement and test new language features, type systems, and runtime behaviors as extensions rather than starting from scratch. Typed Racket, a gradually typed variant, was born directly from this research tradition.
- **Scripting and automation**: Like Python or Ruby, Racket works well for glue code, system administration tasks, data processing, and tooling. The module system keeps things organized as scripts grow into larger projects.
- **Functional programming**: Racket is fundamentally a functional language with first-class functions, closures, immutability by default, and powerful higher-order abstractions. If you value referential transparency and compositional design, Racket delivers.

Racket is less ideal for:

- **Mass-market commercial applications in mainstream stacks**: If your team knows Java, Python, or JavaScript and you are building a standard web application, switching to Racket introduces learning curve and hiring challenges that may not be justified. Racket's ecosystem, while rich, is smaller than those of the dominant languages.
- **Performance-critical systems programming**: Racket compiles to bytecode and optionally uses a JIT compiler on the Chez Scheme backend. It can be fast enough for many tasks, but it will not match C or Rust for raw

performance in compute-intensive applications. Use the FFI to call into native code when you need that level of speed.
- **Mobile development**: There is no mature tooling for deploying Racket applications to iOS or Android. If mobile is your target platform, look elsewhere.
- **Projects requiring mainstream framework ecosystems**: While Racket has web frameworks, ORMs, testing libraries, and more, they are not as numerous or battle-tested as the ecosystems around Django, Rails, Spring, React, or similar frameworks in other languages.

A Brief Tour of What You Will Build

Throughout this book you will write a variety of programs that demonstrate Racket's capabilities:

- Simple calculators and data processors to learn core syntax and functions
- Modular libraries with contracts for safe code organization
- Command-line tools with argument parsing and file I/O
- Web servers serving dynamic content and handling user input
- Database-backed applications using SQLite and PostgreSQL
- Desktop GUI applications with buttons, menus, and event handling
- Concurrent programs using threads, futures, and parallel computation
- Custom macros that introduce new syntactic forms into your code
- A domain-specific language for a specific problem domain
- Optimized, production-quality code with profiling and testing

Each example is complete, runnable, and explained. You can copy the code directly into DrRacket or save it to files and run it from the command line.

How to Read This Book

This book assumes you have programmed in at least one other language and understand basic concepts like variables, functions, conditionals, loops, and data structures. You do not need any prior experience with Lisp, Scheme, or Racket. If you are completely new to programming, consider starting with the free textbook "How to Design Programs" before diving into this book.

Read the chapters in order. Each chapter builds on concepts from earlier ones, and skipping ahead will leave gaps. That said, once you have worked through the core material (chapters 1 through 8), you can jump around to the topics most relevant to your interests. The advanced chapters on macros, language design, concurrency, and performance are largely independent of each other.

All code examples use Racket version 9.3 or later, which is the current stable release as of this writing. If you are using an older version, some features may not be available. When in doubt, check the official documentation at docs.racket-lang.org for version-specific information.

Chapter 1: Origins and Philosophy

From Lisp to Scheme to PLT Scheme to Racket

To understand Racket, you need to understand where it came from. The story begins with Lisp, one of the oldest programming languages still in active use today. Created by John McCarthy in the late 1950s at MIT, Lisp introduced ideas that were revolutionary for their time: code as data, recursive functions, garbage collection, and dynamic typing. These concepts remain central to Racket more than sixty years later.

Lisp spawned two major dialects in the 1970s: Common Lisp and Scheme. Common Lisp grew into a large, feature-rich language with an extensive standard library. Scheme took a different path. Designed by Guy L. Steele Jr. and Gerald Jay Sussman at MIT, Scheme was minimalist and elegant. The original Scheme report fit on just a few pages yet defined a complete programming language with powerful abstractions including lexical scoping, first-class procedures, and tail-call optimization.

Scheme's simplicity made it attractive for teaching and research, but practical software development demanded more: module systems, object-oriented features, better tools, package management. Various Scheme implementations added these features independently, leading to fragmentation. Different Schemes had different libraries, different conventions, and different extension mechanisms. A program written for one Scheme implementation often did not work on another.

In January 1995, a team of researchers at Rice University led by Matthias Felleisen and Matthew Flatt set out to solve this problem. They created PLT Scheme (Programming Languages Toolkit Scheme), a comprehensive development environment that combined a clean Scheme-based language with powerful features for large-scale programming. The first version introduced a unit system for modular programming and a contract system inspired by Bertrand Meyer's design-by-contract methodology.

PLT Scheme grew steadily over the next decade and a half. It added:

- A module system that replaced units, providing cleaner namespace management
- Hygienic macros, preventing accidental variable capture in macro expansions
- Classes and objects for object-oriented programming
- A rich standard library covering networking, databases, GUIs, and more
- DrRacket, an integrated development environment designed for learning and productivity

By 2010 PLT Scheme had evolved so far beyond its Scheme roots that the team felt the name no longer reflected what it was. On June 7, 2010 they announced the renaming to Racket. The change was partly practical (the acronym PLT had long since lost its original meaning) and partly philosophical: Racket was not just another Scheme implementation but a distinct programming language with its own identity.

The latest stable version as of this writing is Racket 9.3, released in August 2026. Version 9.0, released in November 2025, introduced true parallel threads, ending a thirty-year limitation where Racket's concurrency model provided interleaving but not simultaneous execution on multiple cores.

The Homage Report: Designing a Language for Language Design

Racket's design philosophy is captured most clearly in the "Homage Report," a document that explains the principles guiding its development. At its heart is a single idea: Racket should be a platform for programming language design and implementation, not just another general-purpose language.

This means several concrete things:

First, code must be data. In Lisp-family languages this property is called homoiconicity. Programs are represented as lists (or more precisely, S-expressions), which are also the language's primary data structure. This symmetry means that programs can manipulate other programs as easily as they manipulate numbers or strings. It is the foundation for macros and metaprogramming.

Second, the language must be extensible at the syntax level. Most languages let you define new functions and types but not new syntactic forms.

In Racket you can introduce entirely new constructs that look like built-in language features. The `match` form for pattern matching, for example, is not part of the core language but a macro that expands into existing forms. To the programmer it feels like a first-class feature because it behaves like one.

Third, extensions must be safe and composable. Early Lisp macros were powerful but dangerous: they could accidentally capture variables or produce confusing error messages. Racket's hygienic macro system prevents most of these problems by tracking variable bindings automatically. Macros can be composed without stepping on each other's toes because each respects lexical scope.

Fourth, you should be able to create new languages easily. The #lang (hashtag lang) protocol lets any module declare what language it is written in. By default that is Racket itself, but nothing prevents you from defining your own language with its own reader, syntax rules, and semantics. This is how Typed Racket exists alongside untyped Racket: it is simply a different #lang that adds static type checking.

The Homage Report frames these ideas as an "evaluation framework": every design decision in Racket is assessed against whether it supports or hinders language-oriented programming. If a feature makes it harder to create new languages or DSLs, it is suspect regardless of how convenient it might seem for everyday coding.

Homoiconicity and Code as Data

Homoiconicity is one of those terms that sounds intimidating but describes a straightforward idea: in Racket, the representation of a program in source code is itself a data structure of the language. When you write `(add 1 2)`, you are writing a list containing three elements: the symbol `add` and the numbers `1` and `2`. That same list structure can be created programmatically, manipulated, and then evaluated as code.

This property enables several powerful techniques:

Metaprogramming: Since code is data, programs can generate or transform other programs. A macro is essentially a function that takes code (as data) and returns modified code (still as data), which is then compiled and executed. This is how Racket extends its own syntax.

Interpreters and compilers: Writing an interpreter for a language becomes much simpler when the source code is already in a convenient data structure. You parse text into S-expressions, then recursively evaluate them according to your language's rules. Many programming language research projects use Racket as their implementation platform precisely because of this convenience.

Domain-specific languages: When you need specialized syntax for a particular problem domain, you can write a reader that parses your custom syntax into Racket expressions and then evaluates or compiles them. The resulting DSL feels natural to domain experts while leveraging all of Racket's power under the hood.

Here is a simple demonstration of homoiconicity in action:

```
#lang racket

;; This is a normal function call
(add 1 2)

;; But (add 1 2) is also data we can create and manipulate
(define my-code '(+ 1 2))
(my-code)        ;; Not valid: lists are not callable by default

;; We can evaluate it using eval
(eval my-code)   ;; Returns 3

;; Or we can construct code programmatically
(define (make-adder a b)
  (list '+ a b))

(eval (make-adder 5 7))   ;; Returns 12
```

Note that `eval` is rarely needed in well-structured Racket programs. The macro system provides safer, more efficient ways to work with code as data. But this example illustrates the core idea: code and data share the same representation, so they can be freely converted between roles.

The Read-Eval-Print Loop and Interactive Development

Racket inherits from Lisp another defining characteristic: the REPL (Read-Eval-Print Loop). This interactive environment lets you type expressions one at a time, see their results immediately, and iterate rapidly on your code. It is not merely a debugging tool but a primary mode of development.

The REPL works as follows:

1. **Read**: You type an expression and press Enter. The system parses it into an S-expression.
2. **Eval**: The expression is evaluated according to Racket's semantics.
3. **Print**: The result is displayed.
4. **Loop**: The process repeats, with all previous definitions still in scope.

Here is a typical REPL session:

```
Welcome to Racket v9.3.
-> (+ 2 3)
5
-> (define x 10)
-> (* x 4)
40
-> (define (square n) (* n n))
-> (square 7)
49
-> (map square '(1 2 3 4))
'(1 4 9 16)
```

DrRacket, Racket's integrated development environment, extends this model with a split window. The top pane is the definitions area where you write complete programs. The bottom pane is the interactions area (the REPL). When you click Run, DrRacket loads all your definitions into the interactions area, and then you can experiment with them interactively.

This workflow supports exploratory programming: you define functions in the definitions area, test them in the interactions area, refine them based on what you observe, and repeat. It is particularly valuable when learning a new language or tackling unfamiliar problems because you get immediate feedback without the overhead of compiling and running a full program each time.

The command-line REPL is also available via the `racket` executable:

```
$ racket
Welcome to Racket v9.3.
-> (+ 10 20)
30
```

Press Ctrl-C or type `(exit)` to leave the REPL.

When Racket Is the Right Tool

Understanding Racket's philosophy helps you decide when to use it. Here are situations where Racket is an excellent choice:

You need a custom language or DSL: If your problem domain has its own vocabulary and structure, creating a tailored language in Racket can make code more readable, reduce errors, and improve productivity for everyone working on the project. This is Racket's strongest differentiator.

You value rapid development and experimentation: The REPL, concise syntax, and rich standard library let you prototype ideas quickly. If you are exploring algorithms, building internal tools, or iterating on designs, Racket accelerates the process.

You are teaching or learning programming concepts: Racket's gradual language levels and clean semantics make it ideal for education. Students can start with simple constructs and gradually encounter more advanced features without being overwhelmed.

You enjoy functional programming: If you prefer immutable data, pure functions, and compositional design, Racket provides a pristine environment for that style. The language encourages functional patterns while not forcing them.

You are doing research on programming languages: Racket's extensibility makes it a natural platform for implementing and testing new language features. Many academic papers in programming languages use Racket as their implementation substrate.

Here are situations where you might choose something else:

Your team has no Lisp experience and resists learning curves: Racket's parenthesized syntax and different conventions can be jarring for programmers accustomed to C-style languages. While the learning curve is manageable, it is real.

You need maximum raw performance: For compute-intensive applications like high-frequency trading, game physics engines, or large-scale numerical simulation, compiled languages like C++ or Rust will generally outperform Racket. You can mitigate this with the FFI but it adds complexity.

You are building for mobile platforms: There is no mature path to deploying Racket applications on iOS or Android.

Your project depends heavily on mainstream frameworks: If you need the ecosystem of a language like Python (for data science) or JavaScript (for web frontends), switching to Racket may isolate you from valuable tools and community resources.

Racket is not trying to be everything to everyone. It is a focused tool for programmers who value expressiveness, extensibility, and the ability to shape their language to fit their problems. If that resonates with you, read on. The next chapter will get you set up and writing your first programs.

Chapter 2: Getting Started with Racket

Installing Racket on Your System

Installing Racket is straightforward. Visit https://racket-lang.org/download/ and choose the installer for your operating system. Racket provides pre-built installers for Windows, macOS, and Linux (including ARM variants). The installation includes the full language runtime, DrRacket IDE, standard libraries, and command-line tools.

On Windows: Run the downloaded `.exe` installer and follow the prompts. By default it installs to `C:\Program Files\Racket`. Add the Racket `bin` directory to your system PATH if you want to use command-line tools from any terminal.

On macOS: Open the downloaded `.dmg` file and drag the Racket application to your Applications folder. To use command-line tools, either add `/Applications/Racket v9.3/bin` to your PATH or run the helper script provided in the installation directory. The official wiki at https://github.com/racket/racket/wiki/Configure-Command-Line-for-Racket has detailed instructions.

On Linux: Run the downloaded shell script installer. It installs Racket to a directory of your choice (often `~/racket`). Add the `bin` subdirectory to your PATH by adding this line to your `.bashrc` or `.zshrc`:

```
export PATH="$HOME/racket/bin:$PATH"
```

Alternatively, many Linux distributions provide Racket through their package managers, though those versions may lag behind the latest release.

After installation, verify it works by running `racket --version` in your terminal:

```
$ racket --version
Welcome to Racket v9.3.
```

DrRacket: The Integrated Development Environment

DrRacket is Racket's built-in IDE and the recommended starting point for learning the language. Launch it from your applications menu (Windows or macOS) or by running `drracket` from the command line.

The DrRacket window has two main panes separated horizontally:

Definitions area (top): This is where you write your programs. You type complete Racket code here, including function definitions, data structures, and module declarations. Nothing in this pane executes until you click the Run button.

Interactions area (bottom): This is the REPL where expressions are evaluated immediately. After clicking Run, all definitions from the top pane are loaded into this environment, and you can experiment with them interactively. You can also type expressions here without running anything first to explore Racket's behavior.

At the bottom of the window you will see a language selector showing something like `#lang racket`. This specifies which language variant your code is written in. The default `#lang racket` gives you access to the full standard library. For learning purposes, DrRacket also provides restricted language levels (like "Beginning Student Language") that limit available features to help beginners focus on core concepts.

Here is how to write and run your first program:

1. In the definitions area, type:

```
#lang racket
(define (greet name)
  (string-append "Hello, " name "!"))
```

2. Click the Run button (green arrow in the toolbar).
3. In the interactions area, type:

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

4. You should see: `"Hello, World!"`

DrRacket provides several useful features beyond basic editing and evaluation:

- **Syntax highlighting** distinguishes keywords, identifiers, strings, and comments with colors.
- **Parenthesis matching** highlights the matching parenthesis when you click near one, helping you navigate nested code.
- **Error tracing** shows detailed stack traces when your program crashes, including the exact line and context of the error.
- **Stepper** (under the Debug menu) lets you step through your program one expression at a time to see how it evaluates, which is invaluable for understanding recursion and complex transformations.
- **Profiler** helps identify performance bottlenecks in running code.

You can customize DrRacket extensively through Edit > Preferences. Notable settings include enabling automatic parenthesis insertion (which saves typing), choosing a color theme, and configuring key bindings.

The REPL and Interactive Exploration

The REPL is not just for beginners experimenting with syntax. Experienced Racket programmers use it constantly as a primary development tool. Here are some patterns that make REPL-driven development productive:

Exploring functions before using them: When you encounter an unfamiliar function in the documentation, try it in the REPL to understand its behavior:

```
> (string-split "a,b,c" ",")
'("a" "b" "c")
> (filter even? '(1 2 3 4 5))
'(2 4)
> (hash 'name "Alice" 'age 30)
'#hash((age . 30) (name . "Alice"))
```

Developing functions incrementally: Write a function in the definitions area, run it, then test various cases in the interactions area. Fix issues and re-run:

```
;; In definitions area:
(define (sum-list lst)
  (apply + lst))

;; In interactions area after running:
> (sum-list '(1 2 3))
6
> (sum-list '())
0
> (sum-list '(1 -1 2 -2 3))
3
```

Prototyping algorithms: Work out the logic of an algorithm interactively before committing it to a file:

```
> (define data '(10 2 38 4 55 6))
> (sort data <)
'(2 4 6 10 38 55)
> (take (sort data >) 3)
'(55 38 10)
```

Using `require` to load libraries: You can import modules directly in the REPL:

```
> (require racket/list)
> (append '(1 2) '(3 4))
'(1 2 3 4)
```

The command-line REPL works similarly but lacks DrRacket's visual conveniences. Start it with `racket` in your terminal and type expressions directly:

```
$ racket
Welcome to Racket v9.3.
-> (define x 42)
-> (+ x 8)
50
-> (exit)
```

For longer REPL sessions from the command line, consider using `raco repl`, which provides a slightly enhanced experience with better history handling.

Your First Programs: From Hello World to Simple Calculations

Let us write a series of small programs that introduce basic Racket concepts. Each one is complete and runnable in DrRacket or as a standalone file.

Program 1: Hello World

```
#lang racket

(displayln "Hello, World!")
```

The `#lang racket` line declares that this file uses the standard Racket language. Every Racket source file should begin with a `#lang` declaration. The `displayln` function prints its argument followed by a newline. Run this program and you will see:

```
Hello, World!
```

Program 2: A Simple Calculator

```
#lang racket

(define x 10)
(define y 3)

(displayln (format "x = ~a" x))
(displayln (format "y = ~a" y))
(displayln (format "x + y = ~a" (+ x y)))
(displayln (format "x - y = ~a" (- x y)))
(displayln (format "x * y = ~a" (* x y)))
(displayln (format "x / y = ~a" (/ x y)))
```

This program defines two variables and performs arithmetic on them. Key observations:

- `define` creates a named binding for a value.
- Arithmetic operators use prefix notation: the operator comes before its arguments, enclosed in parentheses. This is standard Lisp/Scheme syntax.
- `format` works like printf in C or Python, using `~a` as a placeholder for any value.

Output:

```
x = 10
y = 3
x + y = 13
x - y = 7
x * y = 30
x / y = 3.3333333333333335
```

Program 3: Defining and Using Functions

```
#lang racket

(define (square n)
  (* n n))

(define (cube n)
  (* n (square n)))

(define (sum-of-squares a b)
  (+ (square a) (square b)))

(displayln (format "square of 5 = ~a" (square 5)))
(displayln (format "cube of 3 = ~a" (cube 3)))
(displayln (format "sum-of-squares of 3 and 4 = ~a" (sum-of-squares 3 4)))
```

Functions are defined with `define` followed by a parameter list in parentheses and a body expression. The last expression evaluated becomes the return value. Racket functions always return exactly one value (though multiple values are supported for special cases we will cover later).

Output:

```
square of 5 = 25
cube of 3 = 27
sum-of-squares of 3 and 4 = 25
```

Program 4: Working with Strings

```
#lang racket

(define first-name "Ada")
(define last-name "Lovelace")
(define full-name (string-append first-name " " last-name))

(displayln full-name)
(displayln (string-upcase full-name))
(displayln (string-length full-name))
(displayln (substring full-name 0 3))
```

Racket provides many string manipulation functions. Note that `string-append` concatenates strings, `string-upcase` converts to uppercase, `string-length` returns the character count, and `substring` extracts a portion using start (inclusive) and end (exclusive) indices.

Output:

```
Ada Lovelace
ADA LOVELACE
12
Ada
```

Program 5: Conditionals

```
#lang racket

(define (describe-number n)
  (cond
    [(< n 0) "negative"]
    [(= n 0) "zero"]
    [else "positive"]))

(for ([n '(-5 0 3 10)])
  (displayln (format "~a is ~a" n (describe-number n))))
```

The `cond` form handles multi-way conditionals. Each clause has a test expression and one or more body expressions. Clauses are evaluated in order; the first whose test is true determines the result. The `else` keyword matches anything, serving as a catch-all.

The `for` loop iterates over a list, binding each element to `n` in turn. We will explore iteration patterns in depth later.

Output:

```
-5 is negative
0 is zero
3 is positive
10 is positive
```

Project Structure and File Organization

As your Racket programs grow beyond single files, organizing them into modules becomes essential. The basic principle is simple: each file is a module with its own namespace, and modules communicate through explicit imports and exports.

Here is a minimal project structure:

```
my-project/
├── main.rkt          ;; Entry point
├── math-utils.rkt    ;; Math helper functions
└── string-utils.rkt  ;; String helper functions
```

The library module `math-utils.rkt`:

```
#lang racket

(provide square cube average)

(define (square n)
  (* n n))

(define (cube n)
  (* n (square n)))

(define (average . numbers)
  (/ (apply + numbers) (length numbers)))
```

The `provide` form declares which definitions are exported for other modules to use. Everything not provided remains private to this module.

The main program `main.rkt`:

```
#lang racket

(require "math-utils.rkt")

(displayln (format "square of 6 = ~a" (square 6)))
(displayln (format "average of 10 20 30 = ~a" (average 10 20 30)))
```

The `require` form imports bindings from another module. Using a relative path like `"math-utils.rkt"` refers to a file in the same directory.

Run the project with:

```
$ racket main.rkt
square of 6 = 36
average of 10 20 30 = 20
```

For larger projects, consider organizing code into subdirectories and using Racket's package system (covered in Chapter 8). The key takeaway for now

is: use modules from the start, export only what needs to be public, and keep related functionality grouped together.

With Racket installed, DrRacket running, and your first programs executed, you are ready to dive into the language core. The next chapter explains how Racket code is structured, evaluated, and composed at a fundamental level.

Chapter 3: Expressions, Values, and the Language Core

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

S-Expressions: Parentheses as Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Literals: Numbers, Strings, Characters, Booleans, Symbols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Variables, Definitions, and Bindings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Functions: Definition, Application, and Arguments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

The Evaluation Model: How Racket Computes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 4: Control Flow and Recursion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Conditionals: cond, if, when, unless, and case

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Recursion as the Foundation of Repetition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Tail Calls and Efficient Recursion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Iteration with map, filter, foldl, and foldr

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Pattern Matching for Structural Decomposition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 5: Data Structures and Collections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Pairs and Lists: The Fundamental Building Blocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Vectors for Indexed Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Strings and Characters in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Hash Tables and Sets for Efficient Lookup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Choosing the Right Collection for the Job

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 6: Structured Data, Contracts, and Safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Defining Structures with struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Accessors, Mutators, and Predicates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Contracts: Runtime Guarantees Between Modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Structured Error Handling with Exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 7: Functional Programming in Racket

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Higher-Order Functions: Functions That Take Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Closures and Lexical Scoping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Immutability and Side Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Functional Design Patterns in Racket

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

When to Mutate and When Not To

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 8: Modules, Libraries, and Organization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

The Module System: require and provide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Creating Your Own Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Language Pragmas and #lang Choices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Package Management with raco pkg

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Documentation with Scribble

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 9: Macros and Metaprogramming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

What Macros Are and Why They Matter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Syntax Objects and the Quote/Unquote Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Writing Your First Hygienic Macro

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Pattern Matching in Macro Definitions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Building Domain-Specific Language Constructs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 10: Object-Oriented Programming in Racket

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Classes and Objects: The Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Inheritance and Super Calls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Interfaces and Mixins for Flexible Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

When to Use OOP in Racket

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 11: Concurrency, Parallelism, and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Threads and Lightweight Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Synchronization: Semaphores, Mutexes, and Channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Futures and Parallel Computation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Parallel Threads in Racket 9.0+

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Profiling and Performance Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Optimizing Hot Paths in Racket

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 12: Input/Output, Files, and Networking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Ports and Streams for I/O

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Reading and Writing Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Command-Line Applications with racket/cmdline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Networking Basics: TCP and UDP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 13: Web Development and Databases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Building a Web Server with racket/web-server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Routing and Request Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Templates and Dynamic HTML Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Database Access with DB Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

A Complete Small Web Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 14: GUI Development and Desktop Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

The racket/gui Framework Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Building Windows and Dialogs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Widgets, Layouts, and Event Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

A Small Desktop Application Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

When GUI Development in Racket Makes Sense

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Chapter 15: Language-Oriented Programming and DSLs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

What Is a Language in Racket?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Creating Your Own #lang

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Custom Readers and Syntax Extensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Building a Practical DSL from Scratch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

The Philosophy of Language-Oriented Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Conclusion: Architectural Guidance and Next Steps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Idiomatic Racket Design Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Project Architecture at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Performance and Maintainability Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Deployment and Distribution Options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Further Learning Resources and Community

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

Quick Reference: Key Forms, Functions, and Conventions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringracket.