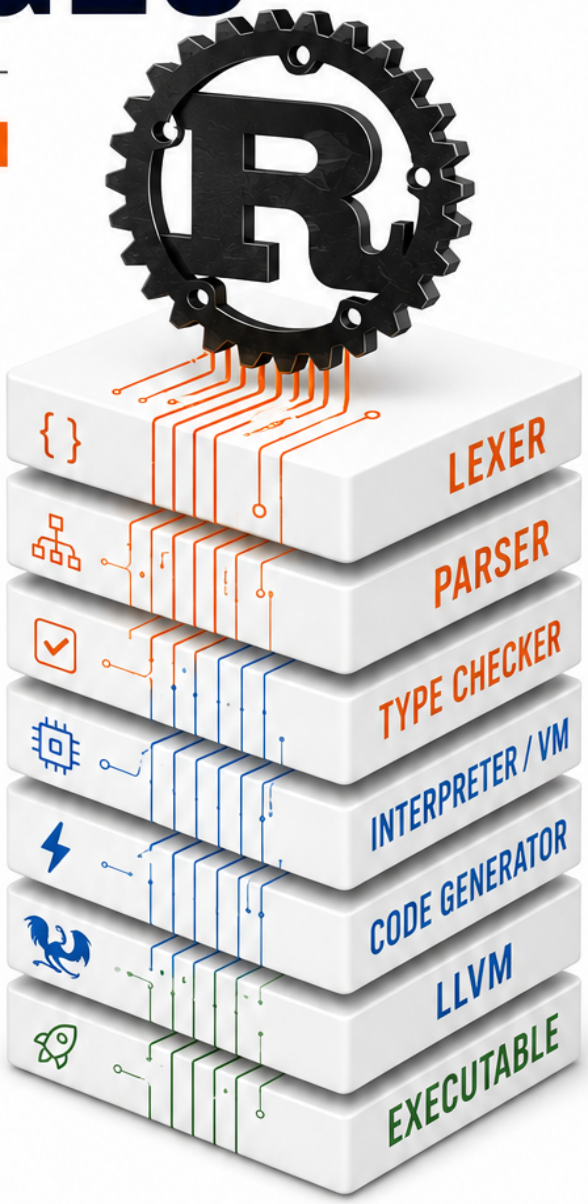
BUILDING PROGRAMMING LANGUAGES WITH RUST



FROM LEXERS TO LLVM: A COMPLETE GUIDE TO LANGUAGE IMPLEMENTATION

- LEXING & PARSING
- TYPE SYSTEMS
- INTERPRETERS & VIRTUAL MACHINES
- NATIVE CODE GENERATION WITH LLVM
- STANDARD LIBRARY
- DEVELOPER TOOLS
  REPL, FORMATTER, LINTER, LANGUAGE SERVER

MATT CONRAD

Building Programming Languages with Rust

From Lexers to LLVM: A Complete Guide to Language Implementation

Steve T. Publications

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

This version was published on 2026-07-14

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 Lexers to LLVM: A Complete Guide to Language Implementation

This book teaches you how to build a complete, production-quality programming language from scratch using Rust. Over twenty chapters, you will design and implement every component of a real language called Rune: the lexer, parser, type system, interpreter, bytecode virtual machine, native code generator, standard library, and full developer tooling chain including a REPL, formatter, linter, and language server. Every chapter includes complete, compilable Rust code with detailed explanations of design trade-offs, performance considerations, and common pitfalls. By the end, you will have built a feature-complete language and deep understanding of how programming languages work internally.

Introduction: The Language Builder's Journey

Every programmer has, at some point, looked at their favorite language and thought: I could make something better. Or perhaps you have encountered a problem that no existing language solves elegantly. Or maybe you simply want to understand what happens between typing code and seeing it run, and the only way to truly know is to build the machinery yourself.

This book is for you.

Why Build a Language?

Building a programming language is the ultimate software engineering project. It touches every major area of computer science: formal language theory, automata, graph algorithms, type theory, memory management, optimization, concurrency, and human-computer interaction. No other single project forces you to confront all of these domains at once.

Consider the alternatives. Building a web application teaches you about HTTP, databases, and user interfaces. Building an operating system teaches you about hardware, scheduling, and resource management. But building a language teaches you about *abstraction itself* – how we represent computation, how we reason about correctness, and how we bridge the gap between human intent and machine execution.

The practical benefits are equally compelling. Understanding language implementation makes you a better programmer in every language you use. You will read error messages differently. You will understand why certain patterns are idiomatic and others are not. You will see through the magic of your favorite features and appreciate the engineering that makes them possible.

Why Rust?

Rust is an exceptional choice for language implementation, and not merely because it is fast. The reasons run deeper.

Memory safety without garbage collection. Compilers and interpreters allocate millions of small objects during a single compilation or execution pass. Traditional approaches either pay the cost of garbage collection pauses or risk subtle bugs in manual memory management. Rust's ownership system gives you the performance of manual management with the safety of automatic collection, and its arena allocation patterns make it trivial to batch-allocate and batch-free the trees that form the heart of any compiler.

Zero-cost abstractions. Language implementations need both high-level data structures (ASTs, type environments, symbol tables) and low-level operations (bytecode emission, register allocation, instruction selection). Rust lets you write the high-level logic without sacrificing the low-level performance. A well-written Rust compiler can match or exceed the speed of C++ implementations while being measurably safer.

A rich ecosystem. The Rust crate ecosystem has mature tooling for every stage of language implementation: `logos` for blazing-fast lexers, `lalrpop` and `pest` for parser generation, `inkwell` for LLVM integration, `tower-lsp-server` for language server protocol support, `reedline` for REPLs, `bumpalo` for arena allocation, and many more. You can stand on the shoulders of giants while still understanding every layer.

Growing real-world precedent. Over 130 programming languages have been implemented in Rust as of 2024, ranging from production systems like the Roc language to research projects and educational tools. The Rust compiler itself (rustc) is written in Rust, demonstrating that the language can bootstrap its own toolchain. Projects like rust-analyzer show how Rust-based language servers can achieve state-of-the-art performance and accuracy.

What Is Rune?

Throughout this book, we will build a real language called **Rune**. It is not a toy calculator or a two-expression DSL. Rune is designed to be a complete, practical programming language with:

- A C-like syntax with modern ergonomics (pattern matching, null safety, modules)
- Static typing with full type inference (Hindley-Milner style)
- First-class functions and closures with lexical scoping
- Algebraic data types and pattern matching
- Traits (interfaces) with both static and dynamic dispatch
- A comprehensive standard library
- An interactive REPL with syntax highlighting and tab completion
- Multiple execution backends: tree-walking interpreter, bytecode VM, and LLVM-based native compilation
- Full developer tooling: formatter, linter, debugger, and language server

You can think of Rune as a cross between TypeScript and Rust: statically typed but ergonomic, systems-capable but accessible. The design decisions we make along the way will be justified by real trade-offs, not arbitrary choices.

How This Book Is Structured

The book is organized into three major phases, each building on the previous one:

Phase 1: From Source to AST (Chapters 1-6). We begin with language design principles, then implement the lexer and parser that transform raw source text into an abstract syntax tree. You will learn formal grammar notation, recursive descent parsing, Pratt parsing for expressions, and arena-based memory management for compiler data structures.

Phase 2: Analysis and Semantics (Chapters 7-9). With a parse tree in hand, we build the semantic analysis layer: symbol tables for name resolution, a Hindley-Milner type inference system, and a professional-grade error reporting engine that produces helpful diagnostics.

Phase 3: Execution and Tooling (Chapters 10-20). We implement three execution backends (interpreter, bytecode VM, LLVM code generator), then build the ecosystem around the language: memory management strategies, optimizations, concurrency primitives, module system, macros, developer tooling, testing infrastructure, and distribution.

Each chapter is self-contained enough to read independently for reference, but the chapters build cumulatively. The Rune language grows feature by

feature, and by the final chapter you will have a complete, working language implementation.

What You Need to Know

This book assumes you are already comfortable with Rust at an intermediate level. Specifically, you should understand:

- Ownership, borrowing, and lifetimes
- Enums, structs, and trait-based polymorphism
- Generics and type parameters
- The standard library (collections, error handling, I/O)
- Basic async/await concepts
- Cargo workspace organization

You do *not* need prior experience with compiler construction or formal language theory. We build all of that from the ground up. If you are unfamiliar with any Rust concept listed above, the official Rust book at doc.rust-lang.org provides excellent coverage before you begin.

How to Use the Code

Every chapter includes complete, compilable Rust code. The code is organized as a Cargo workspace with separate crates for each major component:

```
rune/
  Cargo.toml              # Workspace root
  rune-core/              # Common types, spans, diagnostics
  rune-lexer/             # Tokenizer
  rune-parser/            # Parser (recursive descent + Pratt)
  rune-ast/               # Abstract syntax tree definitions
  rune-semantics/         # Symbol tables, name resolution
  rune-typecheck/         # Type inference and checking
  rune-interpreter/       # Tree-walking interpreter
  rune-bytecode/          # Bytecode VM
  rune-codegen/           # LLVM-based native code generation
  rune-stdlib/            # Standard library
  rune-repl/              # Interactive REPL
  rune-fmt/               # Code formatter
  rune-lint/              # Static analysis linter
  rune-lsp/               # Language server
  rune-cli/               # Command-line interface
```

You can find the complete source code at the book's repository (referenced in the appendices). Each chapter's code builds on previous chapters, so you should follow along sequentially for the best experience. The exercises at the end of each chapter give you opportunities to extend Rune with your own features.

A Note on Scope

Building a complete programming language is a massive undertaking. Professional language teams spend years on individual components. This book compresses that journey into a manageable format by making deliberate trade-offs: we implement the core algorithms from scratch but use well-tested crates where reinventing the wheel would distract from the learning objectives. Where we use external libraries, we explain how they work internally so you understand the machinery beneath.

Some topics receive more attention than others based on educational value rather than production necessity. For example, we implement a full Hindley-Milner type inference system even though many practical languages use simpler approaches, because understanding HM gives you a foundation that makes any type system easier to reason about. Similarly, we build both a tree-walking interpreter and a bytecode VM so you can compare the trade-offs directly.

The Journey Ahead

By the time you finish this book, you will have done something remarkable: you will have built a complete programming language from nothing but source code and ideas. You will understand the intimate details of how code becomes execution, how types become guarantees, and how syntax becomes meaning.

More importantly, you will have developed the intuition to design your own languages for specific domains, to evaluate language trade-offs critically, and to contribute to existing language projects with confidence. The skills you gain here – in formal reasoning, systems design, and performance engineering – will make you a better programmer regardless of what you build next.

Let us begin.

Chapter 1: Why Build a Language? The Landscape and the Vision

What Is a Programming Language, Really?

At its most abstract level, a programming language is a formal system for describing computation. It provides a vocabulary (keywords, operators, identifiers) and a grammar (rules for combining those elements) that together express algorithms in a way both humans and machines can understand. But this definition barely scratches the surface of what a programming language actually *is*.

A practical programming language is an ecosystem. It includes not just the core syntax and semantics but also a standard library, tooling (compiler or interpreter, debugger, package manager), documentation, conventions, and often a community. When programmers talk about "learning Python" or "mastering Rust," they are learning all of these things together.

The implementation of a language – the software that transforms source code into execution – falls along several axes:

Compiler versus interpreter. A compiler translates source code into a lower-level representation (typically native machine code) before execution. An interpreter reads and executes source code directly, translating and running simultaneously. The distinction is not always sharp: many systems compile to an intermediate representation (bytecode) that is then interpreted or just-in-time compiled. Java compiles to bytecode interpreted by the JVM. CPython interprets Python bytecode. V8 JIT-compiles JavaScript to native machine code at runtime.

Ahead-of-time versus just-in-time compilation. Ahead-of-time (AOT) compilers produce executable binaries before the program runs. Just-in-time (JIT) compilers translate code during execution, allowing optimizations based on runtime profiling data. AOT gives faster startup and predictable performance; JIT enables adaptive optimization and platform-independent distribution.

Scripting versus systems languages. Scripting languages prioritize developer productivity: dynamic typing, garbage collection, rich standard libraries, rapid iteration. Systems languages prioritize resource control: manual or deterministic memory management, zero-cost abstractions, direct hardware access. The boundary is increasingly blurred: Rust brings systems-level control to high-level ergonomics; Python's performance has improved dramatically through JIT compilation in projects like GraalPython.

For Rune, we will implement both an interpreter (for rapid development and the REPL) and a compiler backend (via LLVM for native code generation). This gives us the flexibility to use the right tool for each job while understanding both approaches deeply.

The Rust Advantage for Language Implementation

Rust has emerged as a leading choice for language implementation, and the reasons go beyond its reputation for speed. Let us examine the specific advantages in detail.

Ownership and lifetimes eliminate entire classes of compiler bugs. Compilers manipulate complex graph structures: ASTs with cross-references, control flow graphs with back edges, type environments with nested scopes. In C or C++, managing the memory for these structures requires careful bookkeeping that is error-prone at scale. Rust's ownership system ensures that references to AST nodes cannot outlive the arena they were allocated in, eliminating dangling pointer bugs without runtime overhead.

Consider a typical compiler scenario: you parse a file into an AST, then walk the AST to build a type environment, then walk it again to generate code. In C++, you might use shared pointers everywhere (adding reference counting overhead) or raw pointers with manual lifetime management (risking use-after-free). In Rust, you allocate the entire AST in an arena, and the borrow checker guarantees that no reference escapes the arena's lifetime:

```rust
use bumpalo::Bump;

struct Compiler<'arena> {
    arena: &'arena Bump,
    ast: Option<AstNode<'arena>>,
}

impl<'arena> Compiler<'arena> {
    fn parse(&mut self, source: &str) -> Result<&'arena AstNode<'arena>, Error>
    ↪ {
        let ast = self.parse_program(source)?;
        let node = self.arena.alloc(ast);
        self.ast = Some(node);
        Ok(node)
    }
}
```

The lifetime parameter `'arena` on every AST node guarantees that no reference to an AST node can outlive the arena. The compiler's type signature encodes this invariant, and the Rust compiler enforces it at compile time with zero runtime cost.

Iterator-based APIs make tree traversal elegant. Walking an AST or a control flow graph is a fundamental operation in compilers. Rust's iterator system provides a clean, composable way to traverse trees without manual recursion or explicit stack management:

```rust
impl<'a> IntoIterator for &'a AstNode<'a> {
    type Item = &'a AstNode<'a>;
    type IntoIter = AstWalker<'a>;

    fn into_iter(self) -> Self::IntoIter {
        AstWalker { stack: vec![self] }
    }
}
```

Fearless concurrency for parallel compilation. Modern compilers exploit multi-core processors extensively. rustc uses a work-stealing thread pool (rayon) to parse and type-check modules in parallel. Rust's Send and Sync traits make it straightforward to design compiler data structures that can be safely shared across threads, enabling parallel parsing, parallel type checking, and parallel code generation without data races.

The crate ecosystem provides production-ready building blocks. The

Rust language implementation ecosystem is mature and well-maintained. Key crates include:

Crate	Purpose	Downloads (monthly)
`logos`	Procedural macro lexer generator	2M+
`lalrpop`	LR(1) parser generator	500K+
`pest`	PEG parser generator	3M+
`nom`	Parser combinator framework	4M+
`bumpalo`	Bump allocation arena	8M+
`inkwell`	Safe LLVM bindings	200K+
`tower-lsp-server`	LSP server framework	500K+
`reedline`	Readline for REPLs	300K+
`codespan-reporting`	Source-code error reporting	1M+
`proptest`	Property-based testing	2M+

This ecosystem means you can focus on the language-specific logic rather than reinventing infrastructure.

Real-world validation. The Rust compiler itself (rustc) is a production-quality compiler written in Rust, handling one of the most complex type systems in widespread use. rust-analyzer provides IDE support for Rust through the Language Server Protocol and demonstrates how to build incremental analysis systems. The Roc language is a complete functional programming language written in Rust with its own LLVM-based code generator. These projects prove that Rust can handle the full complexity of production language implementation.

Survey of Existing Languages Built in Rust

The landscape of languages implemented in Rust spans from production-ready systems to research prototypes. Understanding what others have built helps us make informed design decisions for Rune.

Roc. Perhaps the most ambitious Rust-based language project, Roc is a purely functional programming language designed for speed and developer happiness. Created by Richard Feldman (known for Elm), Roc compiles to native machine code via LLVM and features principal type inference, managed effects through Tasks, and a platform abstraction that separates application logic from I/O primitives. The Roc compiler is written entirely in Rust and demonstrates sophisticated techniques including a custom garbage collector, whole-program optimization, and an incremental type checker. Roc's design philosophy emphasizes simplicity: a small core language with powerful abstractions built on top.

Carbon. Google's Carbon language, designed as a successor to C++, has its early implementation written largely in C++ but has significant Rust components in its tooling ecosystem. The project demonstrates how modern language projects leverage multiple implementation languages.

Swiftpoint (formerly Swifty). A systems programming language inspired by Swift and Rust, implemented in Rust with LLVM backend. It demonstrates how to build a language that bridges the gap between scripting ergonomics and systems-level performance.

Educational projects. The Rust ecosystem hosts dozens of educational language implementations: toy Lisp interpreters, JavaScript-like scripting languages, domain-specific compilers targeting WebAssembly. These projects collectively form a rich learning resource, and many are cited throughout this book.

The common thread across these projects is that Rust provides the right balance of safety, performance, and developer ergonomics for language implementation. The borrow checker prevents subtle bugs in tree manipulation. The type system catches interface mismatches early. The ecosystem provides building blocks that would take months to implement from scratch.

Designing Rune: Goals and Non-Goals

Every language design involves trade-offs, and being explicit about what you are *not* trying to achieve is as important as defining your goals. Here are Rune's design principles:

Goal: Educational clarity. Every implementation decision in this book serves a pedagogical purpose. We choose algorithms and data structures that are easy to understand and extend, even when more complex alternatives exist. When we discuss a simpler approach first and then optimize it, the progression teaches you how real compiler engineers think.

Goal: Production-quality code. The code in this book is not a toy implementation. It follows Rust best practices: proper error handling with the `thiserror` and `anyhow` crates, comprehensive testing, benchmarking infrastructure, documentation, and idiomatic patterns. You can use Rune's implementation as a starting point for your own language projects.

Goal: Multiple execution backends. By implementing an interpreter, a bytecode VM, and an LLVM-based compiler, you will understand the full spectrum of language execution strategies. Each backend has different trade-offs in startup time, peak performance, memory usage, and implementation complexity.

Goal: A useful standard library. Rune's standard library includes collections (lists, maps, sets), string processing, I/O, math, concurrency primitives, and more. Building a standard library teaches you about API design, documentation, and the boundary between language features and library features.

Non-goal: Turing-complete type system. Unlike System F or dependent type systems, Rune's type system is deliberately decidable. Type inference always terminates, and type checking is efficient. This makes the implementation tractable while still being powerful enough for real programs.

Non-goal: Full Rust compatibility. Rune is not a Rust clone. It borrows ideas from many languages (pattern matching from Rust and OCaml, traits from Rust, syntax from TypeScript and Go) but makes independent choices wherever they improve clarity or ergonomics.

Non-goal: Maximum performance. While we implement optimizations and benchmark thoroughly, Rune is not designed to compete with C or LLVM-optimized languages on raw speed. The interpreter and bytecode VM prioritize

correctness and understandability over peak throughput. The LLVM backend achieves competitive performance for its class of language.

Project Structure and Tooling Setup

Let us set up the Cargo workspace that will house the Rune implementation. A well-organized project structure is essential for a project of this scale, where multiple crates interact in complex ways.

Create the workspace root:

```
# Cargo.toml (workspace root)
[workspace]
resolver = "2"
members = [
    "rune-core",
    "rune-lexer",
    "rune-parser",
    "rune-ast",
    "rune-semantics",
    "rune-typecheck",
    "rune-interpreter",
    "rune-bytecode",
    "rune-codegen",
    "rune-stdlib",
    "rune-repl",
    "rune-fmt",
    "rune-lint",
    "rune-lsp",
    "rune-cli",
]
```

Each crate has a focused responsibility. The dependency graph forms a directed acyclic graph (DAG) where higher-level crates depend on lower-level ones:

```
rune-cli --> rune-repl --> rune-interpreter --> rune-typecheck
                                    |                ^
                                    v                |
                              rune-bytecode <--+     |
                                    |          |     |
                                    v          |     |
                                rune-codegen-+ |     |
                                             | |     |
                                        rune-semantics --> rune-parser -->
                                        ↪  rune-lexer
                                             ^              ^             ^
                                             |              |             |
                                        rune-ast <----------+-------------+
                                             ^
                                             |
                                        rune-core (shared types)
```

The `rune-core` crate provides shared infrastructure used by all other crates: source code positions and spans, diagnostic types for error reporting, and common utility functions. Every other crate depends on it directly or transitively.

Here is the `rune-core` Cargo.toml:

```
[package]
name = "rune-core"
version = "0.1.0"
edition = "2021"

[dependencies]
thiserror = "2"
codespan-reporting = "0.11"
unicode-width = "0.1"
```

The `thiserror` crate provides ergonomic error type derivation. The `codespan-reporting` crate generates beautiful, GCC-style error messages with source code snippets and underlines. The `unicode-width` crate handles the tricky problem of computing the display width of Unicode strings (where some characters occupy two terminal columns).

Let us define the foundational types in `rune-core/src/lib.rs`:

```
//! Core types shared across all Rune crates.
//!
//! This crate provides source location tracking, span management,
//! and diagnostic infrastructure used throughout the language implementation.

pub mod span;
pub mod diagnostics;
pub mod error;

pub use span::{BytePos, FileId, SourceFile, Span};
pub use diagnostics::{Diagnostic, DiagnosticLabel, DiagnosticSeverity};
pub use error::CompilerError;
```

The `Span` type is perhaps the most important data structure in any compiler. It represents a contiguous range of bytes in a source file and is attached to every AST node, every parsed token, and every diagnostic message:

```
// rune-core/src/span.rs
use std::fmt;

/// A unique identifier for a source file.
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct FileId(u32);

impl FileId {
    pub fn new(id: u32) -> Self {
        Self(id)
    }
}

/// A position in a source file, measured in bytes from the start.
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct BytePos(pub u32);

impl BytePos {
    pub fn to_usize(self) -> usize {
        self.0 as usize
    }
}

/// A contiguous range of bytes in a source file.
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct Span {
    pub file: FileId,
    pub start: BytePos,
    pub end: BytePos,
}

```

```rust
impl Span {
    /// Create a span from start and end byte positions.
    pub fn new(file: FileId, start: BytePos, end: BytePos) -> Self {
        Self { file, start, end }
    }

    /// Create a zero-length span (useful for error messages at a single point).
    pub fn dummy() -> Self {
        Self {
            file: FileId::new(0),
            start: BytePos(0),
            end: BytePos(0),
        }
    }

    /// Merge two spans into a span covering both.
    pub fn merge(self, other: Span) -> Span {
        assert_eq!(self.file, other.file);
        Self::new(
            self.file,
            self.start.min(other.start),
            self.end.max(other.end),
        )
    }

    /// Check if this span is empty.
    pub fn is_empty(self) -> bool {
        self.start == self.end
    }

    /// Check if this span contains the given position.
    pub fn contains(self, pos: BytePos) -> bool {
        self.start <= pos && pos < self.end
    }
}

impl fmt::Display for Span {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}..{}", self.start.0, self.end.0)
    }
}
```

Using byte positions rather than line/column positions is a deliberate choice. Line and column numbers require parsing the source text to find newline characters, which adds complexity and overhead. Byte positions are trivial to compute during lexing (just track the current offset). We convert to line/column only when displaying error messages, using a helper function that scans the source text.

The diagnostics module provides the error reporting infrastructure:

```rust
// rune-core/src/diagnostics.rs
use codespan_reporting::{
    diagnostic::{Diagnostic as CodespanDiagnostic, Label},
    files::Files,
    term::{
        self,
        terminal::{ColorSpec, StdoutTerminal, Terminal},
    },
};
use std::fmt;

/// Severity level for a diagnostic message.
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum DiagnosticSeverity {
    Error,
    Warning,
    Note,
    Help,
}

impl From<DiagnosticSeverity> for codespan_reporting::diagnostic::Severity {
    fn from(severity: DiagnosticSeverity) -> Self {
        match severity {
            DiagnosticSeverity::Error => Self::Error,
            DiagnosticSeverity::Warning => Self::Warning,
            DiagnosticSeverity::Note => Self::Note,
            DiagnosticSeverity::Help => Self::Help,
        }
    }
}

/// A label pointing to a specific span in source code.
#[derive(Debug, Clone)]
pub struct DiagnosticLabel {
    pub span: Span,
    pub text: Option<String>,
}

/// A diagnostic message with optional labels and notes.
#[derive(Debug, Clone)]
pub struct Diagnostic {
    pub severity: DiagnosticSeverity,
    pub message: String,
    pub code: Option<String>,
    pub labels: Vec<DiagnosticLabel>,
    pub notes: Vec<String>,
}

```

```rust
impl Diagnostic {
    pub fn error(message: impl Into<String>) -> Self {
        Self {
            severity: DiagnosticSeverity::Error,
            message: message.into(),
            code: None,
            labels: Vec::new(),
            notes: Vec::new(),
        }
    }

    pub fn warning(message: impl Into<String>) -> Self {
        Self {
            severity: DiagnosticSeverity::Warning,
            message: message.into(),
            code: None,
            labels: Vec::new(),
            notes: Vec::new(),
        }
    }

    pub fn code(mut self, code: impl Into<String>) -> Self {
        self.code = Some(code.into());
        self
    }

    pub fn label(mut self, span: Span, text: impl Into<String>) -> Self {
        self.labels.push(DiagnosticLabel {
            span,
            text: Some(text.into()),
        });
        self
    }

    pub fn note(mut self, note: impl Into<String>) -> Self {
        self.notes.push(note.into());
        self
    }
}
```

This diagnostic system will produce output like the following when a type error is detected:

```
error[E0308]: type mismatch
  --> example.rune:5:14
   |
5  |     let x: Int = "hello";
   |              ---   ^^^^^ expected `Int`, found `String`
   |              |
   |              expected this to be `Int`
   |
   = note: strings and integers are not implicitly convertible
```

The builder pattern on `Diagnostic` (methods like `.code()`, `.label()`, `.note()` that return `Self`) makes it easy to construct rich diagnostics throughout the compiler:

```
fn type_check_assignment(lhs: &Assignment) -> Result<(), Diagnostic> {
    let lhs_type = infer_type(&lhs.target)?;
    let rhs_type = infer_type(&lhs.value)?;

    if !types_compatible(&lhs_type, &rhs_type) {
        return Err(Diagnostic::error("type mismatch")
            .code("E0308")
            .label(lhs.target.span, format!("expected `{}`", lhs_type))
            .label(lhs.value.span, format!("found `{}`", rhs_type))
            .note("types must be compatible for assignment"));
    }

    Ok(())
}
```

This error reporting infrastructure, built on `codespan-reporting`, will serve as the foundation for all compiler messages throughout the book. Every parser error, type error, and semantic warning will flow through this system, ensuring consistent, informative feedback for Rune programmers.

Chapter Summary

In this chapter, we established the motivation for building a language in Rust, surveyed the existing ecosystem of Rust-based language implementations, defined Rune's design goals and boundaries, and set up the project structure with shared core types. The `Span` and `Diagnostic` types we defined here will be used throughout every subsequent chapter, from lexer errors to LLVM code generation warnings.

In the next chapter, we dive into language design fundamentals: the principles that guide our syntax choices, semantic model, type system philosophy, and standard library scope. Before writing a single line of parser code, we need to decide what Rune *is* – and those decisions will shape every implementation choice that follows.

Exercises

1. **Explore the workspace.** Create the Cargo workspace structure defined in this chapter. Add minimal `lib.rs` files to each crate member so the workspace compiles. Run `cargo build` from the workspace root and confirm all crates compile successfully.
2. **Span arithmetic.** Extend the `Span` type with methods for: (a) computing the length of a span, (b) checking whether two spans overlap, (c) creating a span from line/column positions given a source file. Write unit tests for each method.
3. **Diagnostic formatting.** Implement a `SourceFiles` struct that implements the `codespan_reporting::files::Files<FileId>` trait, allowing diagnostics to be rendered with actual source code snippets. Test it with a sample error message.
4. **Research comparison.** Pick two languages from the survey section (Roc, Carbon, or any other Rust-implemented language you find). Read their design documentation and write a one-page comparison of their design philosophies, target use cases, and implementation strategies. What would you borrow for Rune? What would you avoid?

Chapter 2: Language Design Fundamentals

Syntax Philosophy: Minimalism Versus Expressiveness

The syntax of a programming language is its face. It is what programmers see every day, and it shapes how they think about the problems they are solving. Good syntax feels invisible – you read code and understand it without thinking about the grammar. Bad syntax forces you to translate mentally, adding cognitive overhead to every line.

There is no universally correct answer for how much syntax a language should have. The design space ranges from minimalism (few keywords, few special forms, maximum flexibility) to expressiveness (rich keyword vocabulary, specialized syntax for common patterns, reduced need for library abstractions).

The case for minimalism. Languages like Python and Go argue that fewer syntactic constructs reduce cognitive load. Python has exactly one way to define a class. Go has no generics until version 1.18 (and even then, the syntax is deliberately simple). The philosophy is: if every programmer uses the same syntax for the same concept, reading someone else's code is effortless. Minimalism also makes the compiler simpler: fewer grammar rules mean fewer edge cases, fewer ambiguity conflicts, and a smaller implementation surface.

The case for expressiveness. Languages like Rust, C#, and TypeScript invest heavily in specialized syntax: pattern matching, async/await, operator overloading, attribute annotations. The argument is that common patterns deserve first-class syntax rather than library workarounds. If half your codebase uses a particular data structure, it should have dedicated syntax for construction and destructuring.

Rune's position. We take a middle ground: the core language has a small set of constructs, but each construct is expressive. Rune avoids syntactic duplication (no separate `while` and `for` when one loop construct suffices) but

provides rich pattern matching and algebraic data types that make common idioms natural.

Here are Rune's syntax design principles, illustrated with examples:

Principle 1: One obvious way. Where Python has multiple ways to create a dictionary (literal, constructor, comprehension, `fromkeys`), Rune provides exactly one. This reduces the "which style should I use?" question that plagues languages with too many equivalent forms.

```
// Rune: one way to create a map
let config = Map {
    "host": "localhost",
    "port": 8080,
    "debug": true,
};
```

Principle 2: Semicolons are optional but consistent. Rune uses newlines as statement terminators (like Python and Go) but allows semicolons for multiple statements on one line or for clarity. This avoids Python's indentation-sensitive parsing while keeping code visually clean.

```
// Both of these are valid Rune
let x = 42
let y = 100

// Also valid: semicolons for single-line grouping
let a = 1; let b = 2; let c = 3
```

Principle 3: Braces, not indentation. While Python's indentation-based syntax is elegant in theory, it creates real-world problems: invisible characters cause bugs, mixing tabs and spaces breaks code, and reformatting tools fight with the grammar. Rune uses braces for blocks, giving you visual structure that survives copy-paste and reformatting.

```
// Rune uses braces
if condition {
    do_something()
} else {
    do_alternative()
}
```

Principle 4: Keywords are reserved. Unlike JavaScript (where `let`, `class`, and `yield` were added later and can conflict with existing variable names), Rune's keywords are always reserved. This simplifies the lexer and parser – every keyword is unambiguously a keyword, never an identifier.

Rune's complete keyword list (32 keywords):

```
fn, let, mut, if, else, while, for, in, return, break, continue
struct, enum, type, trait, impl, use, mod, pub, as
match, true, false, self, Self, super, crate
async, await, try, catch, yield
```

This is comparable to Go (25 keywords) and fewer than Rust (49+ keywords including lifetime markers). The small keyword set makes Rune easy to learn and the lexer simpler to implement.

Core Semantic Model

Syntax defines what programs look like. Semantics defines what they *mean*. The semantic model determines how expressions are evaluated, how variables are bound, and how side effects interact with the rest of the program.

Evaluation strategy: strict (eager) by default. Rune evaluates arguments before calling functions, following the call-by-value strategy used by C, Java, and Rust. This is the most intuitive evaluation model for programmers coming from mainstream languages and makes performance predictable. Lazy evaluation (call-by-need, as in Haskell) enables elegant infinite data structures but adds significant implementation complexity and makes debugging harder because expressions may be evaluated at unexpected times.

```
// In Rune, arguments are evaluated before the function call
let result = max(compute_a(), compute_b())
// Both compute_a() and compute_b() execute before max() is called
```

Call semantics: values are copied or moved. Simple types (integers, booleans, characters) are passed by value – the callee receives a copy. Complex types (structs, enums, collections) follow move semantics: the caller transfers ownership to the callee, and the original binding becomes invalid. This avoids the deep-copy overhead of pass-by-value for large structures while preventing the aliasing bugs of pass-by-reference.

```
let data = Vec::new(1000)  // Allocate a vector with 1000 elements
process(data)               // `data` is moved into process()
// let x = data             // Error: `data` was already moved
```

Side effects are explicit. Rune does not have a pure functional core like Haskell or Roc. Functions can have side effects (I/O, mutation, global state changes), and the type system does not track them. This makes the type inference simpler (no effect polymorphism needed) while still supporting both functional and imperative styles.

```
// Pure function: no side effects
fn square(x: Int) -> Int {
    x * x
}

// Impure function: has side effects (I/O)
fn log_value(x: Int) -> Int {
    println("Value: {}", x)  // Side effect: writes to stdout
    x                        // Returns the value unchanged
}
```

Last expression is the return value. Like Rust, Ruby, and Swift, Rune treats the last expression in a function body as the return value. The `return` keyword is available for early returns but is not required for normal function exit. This eliminates boilerplate and makes simple functions read like mathematical definitions.

```
fn fibonacci(n: Int) -> Int {
    if n <= 1 {
        n
    } else {
        fibonacci(n - 1) + fibonacci(n - 2)
    }
}
// No explicit `return` needed; the last expression is returned
```

Type System Design Choices

The type system is the heart of a statically typed language. It determines what programs are accepted, what errors are caught at compile time, and how much annotation burden falls on the programmer. Rune's type system balances expressiveness with decidability.

Static typing with full inference. Every expression in Rune has a statically determined type, but programmers rarely need to write type annotations. The compiler infers types using a Hindley-Milner-style algorithm, augmented with limited bidirectional checking for cases where pure inference is insufficient (such as overloaded operators or generic function selection).

```
// No annotations needed -- all types are inferred
let numbers = [1, 2, 3, 4, 5]     // Vec<Int>
let names = ["Alice", "Bob"]       // Vec<String>
let pairs = Map {                  // Map<String, Int>
    "Alice": 30,
    "Bob": 25,
}

fn add(a, b) { a + b }            // fn(Int, Int) -> Int
// The compiler infers that a and b must be Int because of the + operator
```

No null. Rune has no null or nil value. Optional values are represented by the `Option<T>` type, which forces programmers to handle the absence case explicitly. This eliminates the "billion dollar mistake" of null pointer dereferences and makes the intent of optional values explicit in the type system.

```
fn find_user(id: Int) -> Option<User> {
    // ... search logic ...
}

let user = find_user(42)
match user {
    Some(u) => println("Found: {}", u.name),
    None => println("User not found"),
}
// Cannot forget to handle the None case -- the compiler enforces it
```

Algebraic data types. Rune supports both product types (structs) and sum types (enums with variants), enabling the algebraic data type patterns that make functional programming powerful. Pattern matching provides ergonomic destructuring of these types.

```
// Product type: all fields present together
struct Point {
    x: Float,
    y: Float,
}

// Sum type: exactly one variant active at a time
enum Shape {
    Circle { radius: Float },
    Rectangle { width: Float, height: Float },
    Triangle { base: Float, height: Float },
}

fn area(shape: Shape) -> Float {
    match shape {
        Shape::Circle { radius } => 3.14159 * radius * radius,
        Shape::Rectangle { width, height } => width * height,
        Shape::Triangle { base, height } => 0.5 * base * height,
    }
}
```

Traits (interfaces) with static dispatch. Rune's trait system is inspired by Rust but simplified. Traits define method signatures that types can implement. By default, trait methods are dispatched statically (monomorphized at compile time for zero overhead). Dynamic dispatch is available through trait objects when runtime polymorphism is needed.

```
trait Drawable {
    fn draw(&self, canvas: &Canvas)
    fn bounds(&self) -> Rect
}

impl Drawable for Circle {
    fn draw(&self, canvas: &Canvas) {
        canvas.draw_circle(self.center, self.radius)
    }
    fn bounds(&self) -> Rect {
        Rect::from_circle(self.center, self.radius)
    }
}
```

No subtyping hierarchy. Unlike Java or C#, Rune has no class inheritance hierarchy. Types do not extend other types. Polymorphism comes exclusively from traits (interfaces), which are explicitly implemented. This avoids the diamond problem, fragile base class issues, and the conceptual complexity of deep inheritance hierarchies. Composition over inheritance is enforced by the type system.

Memory and Ownership Semantics

Memory management is one of the most consequential design decisions for a programming language. It affects performance, safety, implementation complexity, and programmer ergonomics. Rune takes a pragmatic approach that balances these concerns.

The interpreter uses reference counting. When Rune code runs through the tree-walking interpreter or bytecode VM, values are managed by atomic reference counting with cycle detection. This gives deterministic memory reclamation (memory is freed when the last reference drops) while keeping the implementation simpler than a full tracing garbage collector.

```
// Internally, each value on the Rune heap has a reference count
// When the count reaches zero, the value is deallocated immediately
let data = create_large_buffer()
process(data)  // Reference count: +1 for process parameter
// After process returns, reference count drops; if zero, memory freed
```

Cycle detection runs periodically. Reference counting cannot reclaim reference cycles (two objects that reference each other but are otherwise

unreachable). Rune's cycle detector runs when the heap reaches a certain allocation threshold, scanning for and breaking cycles. This avoids the stop-the-world pauses of traditional garbage collectors while preventing memory leaks from cyclic structures.

The LLVM backend uses manual management with RAII. When Rune code is compiled to native machine code via LLVM, memory management follows Rust-like patterns: values are owned and moved, not copied (unless the type implements a `Clone` trait). Smart pointer types (`Rc<T>` for reference counting, `Arc<T>` for thread-safe reference counting) are available in the standard library when shared ownership is needed.

No unsafe keyword. Unlike Rust, Rune does not have an "unsafe" escape hatch that bypasses the type system and memory safety guarantees. All operations are checked at runtime (bounds checks on arrays, null checks on optional values, overflow checks on arithmetic). This simplifies the language specification and eliminates a whole class of vulnerabilities, at the cost of some performance overhead in the interpreter and VM backends. The LLVM backend performs bounds check elimination as an optimization where the compiler can prove safety statically.

The Standard Library Blueprint

A language without a standard library is just a grammar. The standard library determines what programmers can do without reaching for third-party packages, and it establishes idioms that ripple through the entire ecosystem.

Rune's standard library is organized into modules, each with a clear responsibility:

`core` – Fundamental types and functions.

- Primitive types: `Int`, `Float`, `Bool`, `Char`, `String`
- Optional and result types: `Option<T>`, `Result<T, E>`
- Basic traits: `Eq`, `Ord`, `Hash`, `Clone`, `Debug`, `Display`
- Assertions and panics: `assert()`, `panic()`

`collections` – Data structures.

- Sequences: `Vec<T>` (dynamic array), `LinkedList<T>`

- Maps: `HashMap<K, V>`, `BTreeMap<K, V>`
- Sets: `HashSet<T>`, `BTreeSet<T>`
- Queues and stacks: `Queue<T>`, `Stack<T>`

`io` – Input and output.

- Standard streams: `stdin`, `stdout`, `stderr`
- File operations: `File::open()`, `File::write()`, `File::read()`
- Path handling: `Path`, `DirEntry`
- Buffering: `BufferedReader`, `BufferedWriter`

`string` – Text processing.

- String methods: `split()`, `join()`, `trim()`, `replace()`
- Pattern matching: regex support via `Regex` type
- Encoding: UTF-8 handling, character classification

`math` – Mathematical operations.

- Basic: `abs()`, `min()`, `max()`, `clamp()`
- Trigonometry: `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`
- Exponents and logarithms: `pow()`, `sqrt()`, `exp()`, `log()`, `log2()`
- Constants: `PI`, `E`

`time` – Time and dates.

- Clock: `now()`, `sleep()`
- Duration: arithmetic on time intervals
- Formatting: parse and display timestamps

`concurrency` – Parallel execution.

- Threads: `spawn()`, `join()`
- Channels: `Channel<T>` with `send()` and `recv()`
- Synchronization: `Mutex<T>`, `RwLock<T>`, `Barrier`
- Async: `async` blocks, `await`, `Future<T>`

`test` – Testing framework.

- Test declaration: `#[test]` attribute
- Assertions: `assert_eq()`, `assert_ne()`, `assert_matches()`
- Benchmarks: `#[bench]` attribute with timing infrastructure

The standard library is written in Rune itself (bootstrapped from an initial C/Rust implementation), demonstrating that the language can express its own foundational abstractions. This also means that standard library code serves as examples of idiomatic Rune programming.

Chapter Summary

In this chapter, we established Rune's design philosophy across four dimensions: syntax (minimal but expressive, brace-delimited, reserved keywords), semantics (strict evaluation, move semantics, last-expression return), type system (static typing with HM inference, no null, algebraic data types, traits without inheritance), and memory management (reference counting with cycle detection for interpreted mode, ownership-based for native compilation). We also outlined the standard library's module structure.

These design decisions will directly shape the implementation in every subsequent chapter. The choice of strict evaluation simplifies the interpreter. The absence of subtyping makes the type checker more straightforward. Reference counting determines how we manage the VM's heap. Every implementation detail traces back to a design decision made here.

In the next chapter, we begin the implementation proper: formal grammars and lexical analysis. We will define Rune's complete lexical specification, design the token types, and implement a production-quality lexer using both a hand-written state machine and the `logos` procedural macro crate.

Exercises

1. **Design document.** Write a one-page design document for a feature you would like to add to Rune (a new control flow construct, a new data type, a new operator). Justify your design choices using the principles discussed in this chapter. What trade-offs are you making?

2. **Keyword analysis.** Compare Rune's 32 keywords with the keyword sets of Python (35), Go (25), and Rust (49+). What categories of features does each language's keyword set reveal about its design priorities? What Rune keywords might you remove if you wanted a more minimal language? What would you add for a more expressive one?
3. **Type inference thought experiment.** Given the Rune function `fn identity(x) { x }`, what type should the compiler infer? How does this relate to let-polymorphism in Hindley-Milner? Write out the type inference steps manually.
4. **Standard library design.** Design the API for a new standard library module: `rune::net` for network programming. Define at least five types and ten functions/methods. What design decisions do you make about error handling, blocking vs non-blocking I/O, and resource management?

Chapter 3: Formal Grammars and Lexical Analysis

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

Regular Expressions and Token Definitions

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

Building the Lexer: State Machine Approach

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

Error Recovery in Lexing

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

Unicode and Source Encoding

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

Lexer Benchmarks and Optimization

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

Chapter Summary

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

Exercises

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

Chapter 4: Recursive Descent Parsing

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

Context-Free Grammars and Parse Trees

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

AST Node Design in Rust

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

Writing the Parser

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

Expression Parsing and Operator Precedence

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

Error Reporting with Source Spans

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

Chapter Summary

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

Exercises

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

Chapter 5: Advanced Parsing – Pratt Parsers and Parser Generators

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

The Pratt Parser Algorithm

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

Integrating Pratt Parsing into Rune

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

Parser Generators in Rust: LALRPOP Deep Dive

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

Nom and Combinator-Based Parsing

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

Choosing Your Parser Strategy

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

Chapter Summary

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

Exercises

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

Chapter 6: Abstract Syntax Trees and Tree Transformations

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

Arena Allocation for AST Nodes

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

Desugaring Passes

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

Tree-Walking Patterns

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

Source Maps and Debug Information

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

AST Serialization and Caching

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

Chapter Summary

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

Exercises

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

Chapter 7: Semantic Analysis and Symbol Tables

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

Scope Chains and Symbol Resolution

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

Building the Symbol Table

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

Name Mangling and Uniqueness

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

Semantic Error Detection

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

Cross-Module Name Resolution

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

Chapter Summary

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

Exercises

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

Chapter 8: Type Systems and Type Checking

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

Type Representation in Rust

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

The Type Checker Architecture

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

Type Inference with Unification

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

Generics and Monomorphization

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

Trait Resolution and Dispatch

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

Chapter Summary

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

Exercises

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

Chapter 9: Error Reporting and Recovery

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

The Anatomy of a Good Error Message

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

Building the Diagnostics Engine

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

Parse Recovery Strategies

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

Type Error Messages That Help

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

Error Collection and Summarization

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

Chapter Summary

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

Exercises

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

Chapter 10: The Interpreter

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

Evaluation Semantics

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

Environment and Closure Implementation

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

Built-in Functions and Foreign Function Interface

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

Garbage Collection for the Interpreter

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

REPL Implementation

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

Chapter Summary

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

Exercises

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

Chapter 11: Bytecode Virtual Machine

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

Instruction Set Architecture Design

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

Rune's Instruction Set

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

Code Generation from AST to Bytecode

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

The VM Execution Engine

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

Performance: Bytecode vs Tree Walking

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

Chapter Summary

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

Exercises

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

Chapter 12: Native Code Generation with LLVM

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

LLVM Architecture Overview

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

Inkwell: The Rust LLVM Binding

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

Codegen for Expressions and Statements

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

Optimization Passes

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

Linking and Runtime Support

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

Chapter Summary

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

Exercises

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

Chapter 13: Memory Management Strategies

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

Manual Memory Management Model

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

Tracing Garbage Collection in Rust

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

Region-Based Memory Management

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

Ownership and Borrowing for Scripting Languages

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

Memory Profiling and Debugging

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

Chapter Summary

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

Exercises

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

Chapter 14: Optimization Techniques

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

Peephole Optimizations

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

Data Flow Analysis

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

Loop Optimizations

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

Inline Caching and Monomorphization

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

Profiling-Guided Optimization

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

Chapter Summary

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

Exercises

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

Chapter 15: Concurrency Model

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

Threads vs Green Threads vs Actors

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

Rune's Concurrency Primitives

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

Implementing Async/Await

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

Shared-State Concurrency

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

Parallel Compilation

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

Chapter Summary

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

Exercises

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

Chapter 16: Modules, Packages, and the Standard Library

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

Module System Design

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

Package Manager Architecture

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

Building the Standard Library

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

Foreign Function Interface

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

Documentation System

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

Chapter Summary

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

Exercises

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

Chapter 17: Macros and Metaprogramming

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

Macro Design Philosophy

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

Token-Based Macros

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

Hygienic Macro Implementation

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

Compile-Time Evaluation

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

Macro Debugging and Error Reporting

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

Chapter Summary

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

Exercises

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

Chapter 18: Tooling – Formatter, Linter, and Language Server

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

Automatic Code Formatting

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

Static Analysis Linter

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

Language Server Protocol Implementation

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

Debugger Architecture

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

IDE Integration

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

Chapter Summary

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

Exercises

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

Chapter 19: Testing, Benchmarking, and Quality Assurance

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

Test Infrastructure

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

Property-Based Testing for Compilers

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

Fuzzing the Parser and VM

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

Performance Benchmarking

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

Continuous Integration Pipeline

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

Chapter Summary

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

Exercises

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

Chapter 20: Packaging, Distribution, and Maintenance

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

Cross-Platform Compilation

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

Installer and Package Distribution

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

Documentation Strategy

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

Semantic Versioning for Languages

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

Community and Ecosystem Building

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

Chapter Summary

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

Conclusion: The Language Builder's Path Forward

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

References

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

Glossary

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

Appendix A: Complete Cargo Workspace Configuration

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

Appendix B: Rune Language Specification Summary

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

Appendix C: Building and Running Rune

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