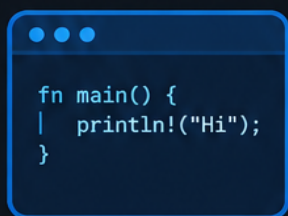


Modern Compiler Engineering

From Source Code to Machine Code

Designing and Implementing a Production
Compiler with Rust



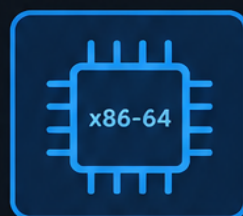
SOURCE CODE



PARSING &
TYPE CHECKING



OPTIMIZATION &
REGISTER ALLOCATION



NATIVE CODE
GENERATION



KIERAN WALLACE

Modern Compiler Engineering: From Source Code to Machine Code

Designing and Implementing a Production Compiler with Rust

Steve Publications

This book is available at <https://leanpub.com/moderncompilerengineeringfromsourcecodetomachinecode>

This version was published on 2026-10-07



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.

© 2026 Steve Publications

Contents

Designing and Implementing a Production Compiler with Rust	1
Introduction	2
From Machine Code to High-Level Languages	3
The Compilation Pipeline in One Picture	4
Compilers, Interpreters and JITs	4
Why Build Your Own Compiler?	5
How This Book Is Structured	6
Chapter 1: What Is a Compiler?	8
The Compiler as Translation Engine	8
The Modular Architecture	8
A Concrete Example	9
Compilers as Complex Systems	11
The Compiler Landscape Today	12
The Lumina Reference Compiler	13
Prerequisites and Conventions	14
The Journey Ahead	15
Chapter 2: Compiler Architecture	17
The Need for Modular Design	17
Phases and Representations	17
The Lumina Pipeline Architecture	18
Error Handling as a First-Class Concern	21
Frontend-IR and Backend Separation	22
The Lumina Build System and Project Layout	23
Design Trade-offs in Compiler Architecture	24
Summary	25
Chapter 3: Language Design for Implementation	27
Language Design and Compiler Complexity	27

CONTENTS

Syntax Design Choices	27
Type System Implications	27
Memory Model and Ownership	27
Evaluation Strategy and Semantics	27
Standard Library and Runtime Expectations	27
The Lumina Language Specification	28
Implementation Language: Why Rust	28
Summary	28
Chapter 4: Lexical Analysis	29
The Role of the Lexer	29
Token Types and Lexical Rules	29
Building a Hand-Written Lexer	29
Understanding the Lexer Code	29
Error Handling in Lexical Analysis	29
Lexer vs. Parser Responsibilities	29
Lexer Engineering and Whitespace Handling	30
Testing the Lexer	30
Summary	30
Chapter 5: Parsing Fundamentals	31
From Tokens to Structure	31
Context-Free Grammars and Syntax	31
LL vs LR Parsing	31
Recursive Descent Parsing	31
Parser Combinators vs Hand-Written	31
Error Recovery in Parsing	31
Implementing Lumina Parse with Recursive Descent	32
Understanding the Parser Code	32
Summary	32
Chapter 6: Abstract Syntax Trees	33
Concrete Syntax Trees vs Abstract Syntax Trees	33
AST Node Design and Representation	33
Type Safety and Pattern Matching	33
Source Location Tracking	33
Desugaring and Pre-Transformation	33
Implementing Lumina AST	33
AST Traversal Patterns	34

Summary	34
Chapter 7: Semantic Analysis and Symbol Tables	35
What Semantic Analysis Does	35
Symbol Tables and Scope	35
Name Resolution Algorithms	35
Declarations and Definitions	35
Shadowing and Visibility Rules	35
Semantic Errors and Diagnostics	35
Implementing Lumina Semantic Analysis	36
Understanding the Semantic Analyzer	36
Summary	36
Chapter 8: Type Systems and Type Checking	37
Static vs Dynamic Typing	37
Type Checking Algorithms	37
Type Inference vs Type Annotation	37
Type Errors and Suggestions	37
Implementing Lumina Type Checker	37
Summary	37
Chapter 9: Intermediate Representations	39
Why Intermediate Representations?	39
Three-Address Code and SSA	39
High-Level vs Low-Level IRs	39
IR Design Principles	39
Implementing Lumina IR	39
Lowering AST to IR	39
Summary	40
Chapter 10: Control Flow Graphs	41
Basic Blocks and Control Flow	41
Building Control Flow Graphs	41
Dominators and Post-Dominators	41
Natural Loops and Loop Headers	41
CFG Optimization	41
Implementing Lumina CFG	41
Summary	42
Chapter 11: Static Single Assignment	43

CONTENTS

SSA and Its Benefits	43
Phi Functions and Versioning	43
SSA Construction Algorithms	43
Sparse Conditional Constant Propagation	43
Optimizing SSA Form	43
Implementing Lumina SSA	43
Summary	44
Chapter 12: Data Flow Analysis	45
Data Flow Analysis Fundamentals	45
Reaching Definitions and Available Expressions	45
Live Variable Analysis	45
Interprocedural Analysis	45
Analysis Precision and Performance	45
Implementing Lumina Data Flow	45
Summary	46
Chapter 13: Core Optimizations	47
The Optimization Framework	47
Dead Code Elimination	47
Constant Propagation and Folding	47
Copy Propagation	47
Common Subexpression Elimination	47
Summary	47
Chapter 14: Loop Optimizations	49
Loop Optimization Opportunities	49
Loop-Invariant Code Motion	49
Strength Reduction	49
Induction Variable Elimination	49
Loop Unrolling and Peeling	49
Implementing Lumina Loop Optimizations	49
Summary	50
Chapter 15: Advanced Optimizations	51
Function Inlining Strategies	51
Escape Analysis	51
Tail Call Optimization	51
Algebraic Simplification	51
Profile-Guided Optimization	51

CONTENTS

Implementing Lumina Inliner	51
Summary	52
Chapter 16: Instruction Selection	53
The Instruction Selection Problem	53
Tree-Pattern Matching	53
Instruction Scheduling	53
Peephole Optimization	53
Lowering for x86-64	53
Implementing Lumina Instruction Selection	53
Summary	54
Chapter 17: Register Allocation	55
Register Allocation Problems	55
Graph Coloring Algorithms	55
Linear Scan Allocation	55
Spilling Strategies	55
Register Pressure	55
Implementing Lumina Register Allocator	55
Summary	56
Chapter 18: Code Generation and Assembly	57
Assembly Language and Encoding	57
Machine Code Generation	57
Relocations and Symbols	57
ABI and Calling Conventions	57
Emitting ELF Object Files	57
Implementing Lumina Code Generator	57
Summary	58
Chapter 19: Linking and Loading	59
The Linker's Job	59
Static vs Dynamic Linking	59
Symbol Resolution and Relocation	59
Link-Time Optimization	59
The Loader and Dynamic Linking	59
Integrating Lumina with the Toolchain	59
Summary	60
Chapter 20: Runtime Support	61

CONTENTS

What Compilers Need from Runtimes	61
Initialization and Finalization	61
Standard Library Design	61
Error Handling and Panics	61
Memory Management Patterns	61
Building Lumina Runtime	61
Summary	62
Chapter 21: Diagnostics and Debugging	63
Error Message Design	63
Source Maps and Location	63
Diagnostic Suggestions	63
Debug Information: DWARF	63
Debugging the Compiler	63
Implementing Lumina Diagnostics	63
Summary	64
Chapter 22: JIT Compilation	65
When and Why JIT?	65
Compilation at Runtime	65
Tiered Compilation	65
Deoptimization	65
LLVM ORC and Modern JIT Infrastructure	65
JIT Design for Lumina	65
Summary	66
Chapter 23: Incremental and Parallel Compilation	67
The Incremental Compilation Problem	67
Incremental Dependency Tracking	67
Query-Based Incremental Systems	67
Parallel Compilation	67
Distributed Compilation	67
Making Lumina Fast	67
Summary	68
Chapter 24: Compiler Testing and Verification	69
Compiler Testing Strategies	69
Unit Testing Compiler Components	69
End-to-End and Regression Tests	69
Compiler Fuzzing	69

CONTENTS

Formal Verification of Compilers	69
Testing Lumina	69
Summary	70
Chapter 25: Performance Engineering	71
Profiling Compiler Performance	71
Compiler Memory Usage	71
Compilation Time Optimizations	71
Cache Efficiency in Compilers	71
Benchmarking the Compiler	71
Profiling Lumina Compilation	71
Summary	72
Chapter 26: Compiler Frameworks and Ecosystems	73
LLVM Architecture and Use Cases	73
GCC: The Veteran Toolkit	73
MLIR: Multilevel Intermediate Representation	73
Cranelfit for WebAssembly	73
Parser Generators: ANTLR, LALRPOP, Tree-Sitter	73
Choosing Your Compiler Stack	73
Summary	74
Chapter 27: Hardware Awareness	75
CPU Architectures and ISAs	75
Instruction Latency and Throughput	75
Memory Hierarchy Awareness	75
Branch Prediction and Control Flow	75
SIMD and Vectorization	75
Hardware Features in Lumina	75
Summary	76
Chapter 28: The Complete Lumina Compiler	77
Architecture Overview	77
Complete Project Structure	77
The Compilation Driver	77
Complete Lumina Program	77
Extending Lumina	77
Summary	77
Chapter 29: Extending and Evolving a Compiler	79

Adding Language Features	79
Supporting New Platforms	79
Scaling the Codebase	79
Maintaining Compiler Quality	79
Compiler Versioning and Breaking Changes	79
Community and Ecosystem	79
Summary	80
Conclusion	81
References	82

Designing and Implementing a Production Compiler with Rust

This book takes you through the complete process of designing and implementing a modern compiler from the ground up. You will build Lumina, a statically typed, compiled language with a full frontend, optimizing intermediate representation, register allocator and native code generator for x86-64. Along the way, you will learn the architectural decisions, engineering trade-offs and practical techniques used by production compilers like Clang, GCC, rustc and LLVM. Every chapter includes complete, working code examples in Rust that you can compile, extend and study. By the end, you will not only understand how compilers work, you will know how to build, optimize, test and evolve one in practice.

Introduction

Every line of code you write in a high-level language is the result of decades of compiler engineering. When you compile a Rust program, you are invoking a system that transforms source text into a hierarchical abstract syntax tree, lowers it through multiple intermediate representations, performs dozens of optimization passes, allocates a handful of hardware registers and emits machine instructions that match the calling conventions of your operating system. The compiler must be correct enough that your program does not miscompile, fast enough that you can iterate quickly and robust enough to handle millions of lines of real-world code without crashing.

This book is about understanding that system and building one yourself.

The reference compiler we will build together is called Lumina. It is a statically typed, imperative language with C-like syntax, value semantics and a simple ownership model. Lumina is not designed to compete with Rust or Zig or Go. It is designed to be complete enough to demonstrate real compiler engineering while remaining small enough to fit in a single book. By the final chapter, Lumina will have:

- A hand-written lexer that tokenizes source code and reports precise errors
- A recursive descent parser that builds a typed abstract syntax tree
- A semantic analyzer with symbol tables and name resolution
- A type checker with type inference and precise error diagnostics
- A custom intermediate representation in SSA form
- An optimizer with dead code elimination, constant propagation and loop optimizations
- A register allocator and instruction selector for x86-64
- Integration with the system linker to produce native executables
- A runtime library for standard operations and error handling

We will implement Lumina in Rust. Rust is a natural choice for compiler construction: its strong type system catches bugs in compiler code at compile

time, its performance characteristics match the demands of large-scale compilation and its growing ecosystem provides excellent tools for the job. But the principles you will learn are transferable to any implementation language.

This book prioritizes engineering over theory where the two diverge. You will learn about LR parsing and LL parsing because they explain why production parsers are built the way they are. You will learn about data flow analysis lattices because they are the theoretical foundation of every optimization pass. But you will also learn about project structure, incremental compilation, error reporting and compiler testing: the practical concerns that separate a working prototype from a tool engineers actually use.

Let us begin with the big picture.

From Machine Code to High-Level Languages

Computers only execute machine instructions: binary patterns that tell the CPU to load a value from memory, add two numbers, or branch to another instruction. In the early days of computing, programs were written directly in machine code, with each binary pattern corresponding to a specific operation on a specific processor. This was error-prone and nearly impossible to maintain.

Assembly language was the first major abstraction, introducing mnemonic names for machine instructions and symbolic names for addresses. Assembly was a breakthrough, but it remained tied to a specific processor architecture and required programmers to manage every register and memory location explicitly.

The first real compilers emerged in the 1950s. Konrad Zuse envisioned automatic translation for his Plankalkül language as early as 1942, but it was IBM's FORTRAN compiler, developed by John Backus and released in 1957, that proved compilers could produce code competitive with hand-written assembly. This was a watershed moment: for the first time, programmers could describe what they wanted in terms close to mathematics and let the compiler figure out how to make the CPU do it.

The 1960s saw an explosion of compiler innovation. COBOL demonstrated cross-platform compilation, running on multiple machine architectures. ALGOL introduced block structure and nested scopes, requiring more sophisticated semantic analysis. Donald Knuth invented LR parsing in 1965 and

Frances E. Allen pioneered data flow analysis and optimization techniques at IBM that remain foundational today. The Production Quality Compiler-Compiler (PQCC) project in the late 1970s formalized the modular compiler architecture we still use: a front end that parses and checks code, a middle end that optimizes and a back end that generates machine instructions.

The Compilation Pipeline in One Picture

A modern compiler is not a single monolithic program, but a pipeline of transformations. Each phase takes a representation of the program as input and produces a new representation as output, progressively moving from high-level abstractions toward machine-specific details.

The typical pipeline looks like this:

- Source code (text) is lexed into tokens by the lexer
- Tokens are parsed into an abstract syntax tree by the parser
- The semantic analyzer resolves names, builds symbol tables and checks scopes
- The type checker verifies types and infers missing type annotations
- The compiler lowers the AST into an intermediate representation (IR)
- The IR is converted into a control flow graph and then into SSA form
- Optimization passes transform the SSA IR: dead code elimination, constant propagation, loop optimizations
- The optimized IR is lowered to target-specific instructions
- The register allocator maps virtual registers to physical CPU registers
- Machine code is emitted, either as assembly text or binary object files
- The linker combines object files and libraries into an executable
- The loader maps the executable into memory and starts execution

Each phase is designed to be independent of the others, separated by well-defined interfaces. The parser does not care whether the source language is C or Rust. The optimizer does not care which front end produced the IR. The code generator does not care what optimizations were performed. This modularity is what makes large compiler frameworks like LLVM practical: you can add a new front end without touching the optimizer, or add a new optimization without breaking any existing language support.

Compilers, Interpreters and JITs

Compilation is not the only way to run a program. Interpreters execute source code or bytecode directly, translating each instruction at runtime. This approach gives fast startup times and interactive development, but execution is typically much slower than compiled code.

Just-in-time (JIT) compilation attempts to get the best of both worlds. A JIT compiler translates code to native machine instructions during program execution, using runtime profiling information to guide optimization decisions. The JVM uses tiered compilation: code starts interpreted, then gets compiled by a fast baseline compiler (C1) and eventually gets recompiled by an aggressive optimizing compiler (C2) that uses accumulated profiling data. JavaScript engines like V8 use similar techniques.

JIT compilation is fundamentally different from ahead-of-time (AOT) compilation. The JIT sees which code paths are actually taken, which allows it to optimize hot paths aggressively and skip dead code entirely. It can also make speculative optimizations based on runtime observations: assuming a variable always has a certain type, specializing method calls for known types, eliminating redundant checks. If a speculation fails, the JIT deoptimizes: it reverts to slower but correct code, or recompiles with different assumptions.

AOT compilers cannot make these runtime-based optimizations, but they produce executables that start instantly, run deterministically and can be distributed as standalone binaries. Most systems programming languages use AOT compilation because predictability and startup time matter more than the marginal gains from runtime profiling.

Why Build Your Own Compiler?

You might reasonably ask why you would want to build a compiler in an era when excellent compiler frameworks already exist. The answer depends on your goals.

If you want to write better code, understanding compilation helps you write code that the compiler can optimize effectively. You will understand why loop structure affects vectorization, why certain data layouts are cache-friendly, why inlining decisions affect optimization opportunities and why

some patterns compile to better code than others. This knowledge is invaluable for performance-critical work.

If you want to build a domain-specific language, compiler knowledge is essential. DSLs appear everywhere: query languages, configuration formats, build systems, shader languages, scripting languages embedded in larger systems. A DSL compiler is fundamentally a small compiler and the same principles apply.

If you want to contribute to an existing compiler, you need to understand how it works. LLVM has over half a million lines of C++. Contributing effectively requires understanding the intermediate representation, the optimization passes and the target descriptions.

And if you simply want to understand how software works at a fundamental level, compiler construction is one of the best ways to do it. A compiler touches every layer of computing: formal language theory, algorithms and data structures, memory management, CPU architecture, operating systems and toolchain integration. Building one forces you to understand how these pieces fit together.

How This Book Is Structured

The book is organized as a progressive build of the Lumina compiler, with each chapter adding new capabilities and explaining the underlying concepts in detail.

The first part covers the front end: language design, lexical analysis, parsing, abstract syntax trees, semantic analysis and type checking. You will learn how to design a language with compiler implications in mind, how to build a robust lexer and parser and how to implement name resolution and type checking that produces clear, actionable error messages.

The second part covers intermediate representations and optimization. You will design an IR suitable for optimization, convert it to SSA form, implement data flow analysis and build optimization passes that transform the IR while preserving semantics. This is where the compiler does most of its magic.

The third part covers the back end: instruction selection, register allocation and code generation. You will learn how to map the abstract IR to concrete CPU instructions, how to assign a limited set of registers to many variables and how to emit object files that the system linker can combine into an executable.

The fourth part covers advanced topics and ecosystem integration: JIT compilation, incremental compilation, debugging information, compiler testing, performance engineering and the major compiler frameworks in use today. You will see how production compilers solve problems at scale and how to choose the right tools for your own project.

Throughout, the code examples are complete and working. You can download the Lumina compiler code at each stage, run it, modify it and experiment. The book assumes familiarity with programming concepts, basic data structures and low-level concepts like memory layout and calling conventions. It does not assume prior experience with compiler construction.

The goal is not to teach you academic compiler theory. The goal is to give you the practical understanding and engineering skills to build, understand and evolve a real compiler.

Chapter 1: What Is a Compiler?

The Compiler as Translation Engine

At its core, a compiler is a translation engine that transforms a program written in one language into an equivalent program in another language. The source language is typically a high-level language designed for human programmers: expressive syntax, rich abstractions, platform independence. The target language is typically machine code designed for hardware: concrete instructions, specific registers, architecture-dependent details.

The key word is equivalent. A correct compiler preserves the semantics of the source program: the compiled program must compute the same results as the source program for all valid inputs. If your source program prints “hello world” and exits with code 0, the compiled program must do exactly that. If your source program sorts an array, the compiled program must produce the same sorted array.

Semantic preservation is a mathematical property, but practical compilation is an engineering problem. You are translating from one representation of computation into another and every representation has its own constraints and opportunities. The source language might allow arbitrary-precision integers, but the target hardware only has 64-bit registers. The source language might use garbage collection, but the target hardware has no concept of memory reclamation. The compiler must reconcile these differences: implementing missing features through runtime support, raising errors when translation is impossible, or silently mapping to compatible operations when safe.

The Modular Architecture

Modern compilers are modular by design. The three-phase architecture consists of:

1. Front end: Language-specific phases that understand source code. Includes the lexer, parser, semantic analyzer and type checker. The front

end takes source text and produces a language-independent intermediate representation.

2. Middle end: Language-independent optimization phases. The optimizer takes the intermediate representation and applies transformations that improve performance, reduce size, or enable further optimizations. Optimizations include dead code elimination, constant propagation, loop transformations, function inlining and many others.
3. Back end: Target-specific phases that understand hardware. The back end takes optimized IR and generates machine code for a specific processor architecture. Includes instruction selection, register allocation, instruction scheduling and code emission.

This architecture is not arbitrary. It emerges from the observation that many compilation tasks are language-independent or target-independent. Once you have parsed code into a standard intermediate representation, you can run the same optimizations regardless of whether the source was C, Rust, or Swift. Once you have optimized IR, you can generate code for x86, ARM, or RISC-V using the same basic algorithms, differing only in instruction selection and register allocation.

The LLVM project made this architecture famous, but it predates LLVM by decades. GCC uses a similar three-phase design with its own intermediate representations. Even single-language compilers like `rustc` benefit from modular architecture: the type checker can be developed independently of the code generator and optimizations can be improved without breaking any front end.

A Concrete Example

Let us follow a single function through the compilation pipeline to see what actually happens. Consider this simple C function:

```
1  int square(int x) {  
2      int result = x * x;  
3      return result;  
4  }
```

When Clang compiles this function, the following transformations occur:

The lexer reads the source text and produces tokens: `int`, `square`, `(`, `int`, `x`, `)`, `{`, `int`, `result`, `=`, `x`, `*`, `x`, `;`, `return`, `result`, `;`, `}`. Whitespace and newlines

are discarded. Each token includes its type (keyword, identifier, operator) and source location for error reporting.

The parser consumes the tokens and builds an abstract syntax tree. The tree encodes the program structure: a function declaration named “square” with one parameter of type `int`, containing a function body with a variable declaration and a return statement. The parser checks that parentheses match and that the syntax follows C grammar rules.

The semantic analyzer builds a symbol table and resolves names. It records that “`x`” is a parameter of the function and “`result`” is a local variable. It verifies that “`x`” is declared before it is used in the expression `x * x`.

The type checker verifies types. It confirms that `x * x` is an integer expression (integer times integer produces integer), that the assignment to `result` is valid (integer assigned to integer) and that the return statement returns an integer matching the function’s return type.

The compiler lowers the AST to LLVM IR. For this function, the IR looks approximately like this:

```
1  define i32 @square(i32 %x) {  
2    %result = mul i32 %x, %x  
3    ret i32 %result  
4  }
```

The IR is a strongly typed, three-address intermediate language. Each operation produces at most one result. Types are explicit: `i32` means a 32-bit signed integer. The LLVM IR closely mirrors the semantics of the source but in a form that is easier to optimize and translate.

The optimizer runs its passes. With optimization enabled, it notices that `result` is a temporary variable used only once and eliminates it through copy propagation and dead code elimination. The optimized IR becomes:

```
1  define i32 @square(i32 %x) {  
2    %1 = mul i32 %x, %x  
3    ret i32 %1  
4  }
```

The optimizer also recognizes this as a hot candidate for inlining: if this function is called frequently, replacing the call with `x * x` directly could save function call overhead.

The back end maps the optimized IR to x86-64 instructions. Using the System V AMD64 calling convention, the argument *x* arrives in the RDI register and the return value goes in RAX. The code generator produces:

```
1      .intel_syntax noprefix
2      .globl square
3 square:
4      imul rdi, rdi    # rdi = rdi * rdi
5      mov rax, rdi    # return value in rax
6      ret
```

Two instructions. The multiply instruction overwrites RDI with the product. The result is copied to RAX for the return value. The `ret` instruction returns to the caller.

The object file encoder wraps this assembly into an ELF object file with proper sections, symbol tables and relocation entries. The linker later combines this object file with other objects and libraries to produce the final executable.

This example is deliberately simple, but the same pipeline applies to programs of any size. A million-line program goes through the same stages: lexing, parsing, semantic analysis, type checking, IR generation, optimization, code generation and object file emission. The complexity lies in handling large programs efficiently, reporting errors clearly and generating optimized code that performs well on real hardware.

Compilers as Complex Systems

A production compiler is not just an algorithm: it is a complex software system with non-trivial requirements across multiple dimensions.

Correctness is paramount. A compiler bug can introduce subtle errors into every program it compiles. The CompCert verified C compiler addresses this by providing a machine-checked proof that its output is semantically equivalent to its input for almost all of ISO C 2011. Most compilers rely on extensive testing instead: unit tests for individual components, regression tests for known bugs, fuzzing to discover edge cases and comparison against reference implementations.

Performance matters on two axes. Compilation time directly affects developer productivity. Rust’s compilation times are often discussed in the community because slow compilation slows iteration. Memory usage is critical: large projects may require gigabytes of memory to compile and memory-intensive compilation can slow the entire system. The LLVM project has invested significantly in reducing compilation time and memory usage, with features like parallel code generation and incremental compilation.

Runtime performance of the generated code is the most visible measure of compiler quality. Better optimization means faster programs. Profile-guided optimization uses runtime data from test runs to guide optimization decisions, typically improving performance by several percent. Link-time optimization allows whole-program analysis, enabling aggressive inlining and dead code elimination across compilation units.

Error reporting is a user experience problem as much as a technical one. A compiler that reports “error at line 42: unexpected token” is not helpful. A good compiler tells the programmer what went wrong and how to fix it. The Rust compiler invests heavily in error messages, providing suggestions for fixing typos, explaining lifetime errors and suggesting alternative syntax. Clear error messages reduce development time more than any other single compiler feature.

Extensibility allows the compiler to grow with the language. Adding a new feature to a language requires changes to the parser, type checker, optimizer and code generator. A modular architecture with clear interfaces makes these changes manageable. LLVM’s design philosophy prioritizes forward progress over backward compatibility to prevent design calcification: interfaces can evolve as long as existing code continues to work.

Portability is essential for broad adoption. A compiler that only works on Linux is limited. LLVM targets dozens of architectures: x86, x86-64, ARM, AArch64, RISC-V, PowerPC, SystemZ, WebAssembly and many others. Each target has its own instruction set, calling conventions and assembly syntax. LLVM’s target description system, based on declarative TableGen files, allows adding new architectures without rewriting core infrastructure.

The Compiler Landscape Today

The modern compiler landscape is dominated by a few major frameworks.

LLVM is the most widely used compiler infrastructure. It provides a complete pipeline from front end to back end. Clang is LLVM's primary front end, compiling C, C++, Objective-C and OpenCL. LLVM is also the backend for Rust, Swift, Julia, Kotlin and many others. The LLVM project includes tools like lld (linker), libc++ (C++ standard library), MLIR (multi-level IR) and Flang (Fortran). Industry adoption is extensive: Apple uses LLVM as the core of Xcode and companies like Intel, AMD, Nvidia and Sony rely on it for their toolchains.

GCC remains important, especially in embedded systems and the Linux ecosystem. Created by Richard Stallman in 1987, GCC pioneered many compiler technologies including tree-SSA optimization and link-time optimization. It uses three intermediate representations: GENERIC, GIMPLE (SSA-based) and RTL (Register Transfer Language). GCC supports an even wider range of languages than LLVM: Ada, C, C++, D, Fortran, Go, Java, Objective-C and more.

Specialized compilers serve specific domains. The Java HotSpot JVM uses JIT compilation with tiered optimization. The Go compiler, written in Go itself, prioritizes fast compilation times over maximum optimization. The Zig compiler is notable for its clean architecture, single-pass compilation design and integrated linker. WebAssembly compilers target the portable binary format, with Cranelift providing a fast, secure compiler backend used by Wasmtime.

The Lumina Reference Compiler

Lumina, the compiler we will build, is designed to illustrate production-quality compiler engineering through concrete implementation. It is not a toy compiler that only handles arithmetic expressions. It is not a research compiler that implements experimental ideas without practical concern. Lumina is a working, usable compiler that demonstrates how real compilers are built.

The Lumina language specification is kept deliberately simple:

- C-like syntax with familiar constructs: functions, variables, control flow, expressions
- Static typing with basic types: integers, floats, booleans, pointers, structs
- Value semantics: variables hold values directly, not references
- Simple ownership model: no garbage collection, no automatic reference counting

- Procedures return at most one value
- Arrays and structs for composite data
- Standard library for basic operations: printing, memory allocation, I/O

Lumina does not have:

- Complex type system features: generics, traits, pattern matching, dependent types
- Concurrency primitives: threads, channels, `async/await`
- Metaprogramming: macros, templates, reflection
- Object-oriented features: classes, inheritance, polymorphism

These omissions are deliberate. They reduce complexity so we can focus on compiler engineering rather than language design. Every feature Lumina has is there to teach something about compilation.

The Lumina implementation is written in Rust. The codebase is organized into crates:

- `lumina_lexer`: Tokenization and lexical analysis
- `lumina_parser`: Parsing and AST construction
- `lumina_semantic`: Name resolution and symbol tables
- `lumina_type`: Type checking and type inference
- `lumina_ir`: Intermediate representation and SSA form
- `lumina_opt`: Optimization passes
- `lumina_backend`: Instruction selection, register allocation, code generation
- `lumina_driver`: The main compiler executable that orchestrates the pipeline

Each crate is designed to be independently testable and usable. The driver integrates them into a complete compilation pipeline, similar to how real compilers like `rustc` and `Clang` are organized.

As we progress through the book, we will implement each component in detail. The code will grow incrementally, with each chapter adding new capabilities. By the end, you will have a complete compiler that accepts Lumina source code and produces working native executables.

Prerequisites and Conventions

This book assumes familiarity with the following:

- Programming in a systems language: C, C++, Rust, or similar. You should be comfortable with pointers, memory layout and low-level operations.
- Basic data structures and algorithms: hash tables, trees, graphs, sorting, search.
- Computer architecture concepts: CPU registers, instruction sets, memory hierarchy, stack and heap.
- Operating system concepts: processes, memory mapping, loading and linking.
- Familiarity with command-line tools and build systems.

The book does not assume:

- Prior experience with compiler construction
- Knowledge of formal language theory
- Understanding of SSA form or data flow analysis

These concepts will be explained from first principles as needed.

Code examples use Rust. The Rust edition targeted is 2021, with features from stable releases through 2026. The code should compile with any recent Rust toolchain, though very old versions may require adjustments. All examples are tested and verified against current implementations.

Source locations in the book refer to the Lumina compiler repository structure. You can find the complete source code at each stage of development. The repository is organized chronologically by chapter so you can see the compiler evolve.

The Journey Ahead

Over the next twenty-eight chapters, we will build the Lumina compiler from nothing. We will start with language design decisions and implement a lexer that transforms text into tokens. We will build a parser that recognizes valid programs and rejects invalid ones. We will implement semantic analysis

that resolves names and checks scopes. We will design a type system and implement type checking with helpful error messages. We will create an intermediate representation and transform programs into SSA form. We will implement optimization passes that improve code quality. We will build a register allocator and code generator that emit machine instructions. We will integrate with the system linker to produce real executables. We will discuss JIT compilation, incremental compilation, testing strategies and the major frameworks used in the field today.

Each chapter focuses on a specific aspect of the compiler, but they build on each other. Early chapters establish foundations that later chapters depend on. If you are implementing your own compiler alongside the book, keep all the code. Each phase depends on the phases before it.

If you are reading without implementing, the principles still apply. The same architectures, algorithms and trade-offs appear in every compiler. Understanding Lumina will help you understand rustc, Clang, GCC, the JVM and any other compiler you encounter.

The field of compiler engineering has matured over seventy years of innovation, from the first FORTRAN compiler to the modern frameworks that power today's programming languages. The fundamental ideas remain the same: transform code through well-defined phases, preserve semantics at each step, optimize aggressively, generate efficient machine code and report errors clearly. The tools have improved dramatically. The principles have not.

Let us begin.

Chapter 2: Compiler Architecture

The Need for Modular Design

A compiler that compiles one language to one target is conceptually simple: read source, check it, optimize it, emit code, done. The complexity arises when you want to support multiple languages, multiple targets, multiple optimization levels and all the features a production compiler needs. A monolithic architecture that handles everything in one giant program becomes unmaintainable quickly.

Modular design solves this by dividing the compiler into independent components with clear interfaces. Each component does one thing well and communicates with other components through well-defined data structures. Changes to one component should not require changes to others. This is not just theoretical elegance: it is practical necessity for compilers that grow to hundreds of thousands of lines of code.

The LLVM project is the most influential example of successful modular compiler design. Chris Lattner designed LLVM as a set of reusable libraries rather than a monolithic compiler executable. This decision enabled LLVM to become the backbone for dozens of language compilers and toolchains. Clang compiles C/C++ to LLVM IR. rustc compiles Rust to LLVM IR. Swift uses LLVM for code generation. The same optimizer and backend infrastructure serves all of them. When a new optimization is added to LLVM, every language that uses LLVM benefits.

Phases and Representations

A compiler is a sequence of phases, each transforming one representation of the program into another. The representations are the key architectural element: they define what information each phase sees and what it can do.

At the highest level, there are three categories of representations:

Source-level representations describe the program in terms close to the original source code. These include:

- The source text itself: raw characters
- The token stream: lexical units like keywords, identifiers, literals
- The concrete syntax tree: complete parse tree including all syntactic details
- The abstract syntax tree: simplified tree structure without syntactic noise
- The high-level IR: typed, semantically analyzed representation close to source

Intermediate representations are the workhorse of compilation. They abstract away from both source syntax and target hardware, enabling language-independent analysis and optimization. Examples include LLVM IR, GCC's GIMPLE and Cranelift's CLIF. A good IR:

- Is unambiguous: every program construct maps to exactly one IR representation
- Is rich enough to represent important program properties: types, control flow, data dependencies
- Is simple enough to analyze and transform efficiently
- Has well-defined semantics so transformations are provably correct

Target-level representations describe the program in terms specific to the target hardware. These include:

- The target-specific IR: instructions and registers for a specific architecture
- The assembly text: human-readable machine instructions
- The machine code: binary encoding of instructions
- The object file: relocatable binary with sections and symbols

Each representation is designed for specific tasks. The AST is good for capturing program structure and source locations. The IR is good for analysis and transformation. The assembly is good for debugging and verification. Using the wrong representation for a task causes unnecessary complexity.

The Lumina Pipeline Architecture

Lumina uses a multi-phase pipeline with distinct separation between front end, middle end and back end. Let us examine each phase in detail.

Phase 1: Lexical Analysis Input: Source text (a string of characters) Output: Token stream (a sequence of typed tokens)

The lexer reads the source text character by character, groups characters into tokens based on lexical rules and discards whitespace and comments. Each token has a type (keyword, identifier, literal, operator), a value (the actual text or parsed value) and a source location (file, line, column). The lexer does not understand grammar or semantics: it only recognizes patterns.

For example, given the source text `let x = 42;`, the lexer produces:

- Token(Keyword(let), span: 0..3)
- Token(Identifier(x), span: 4..5)
- Token(Eq, span: 6..7)
- Token(IntLiteral(42), span: 8..10)
- Token(Semi, span: 10..11)

Phase 2: Parsing Input: Token stream Output: Abstract Syntax Tree

The parser consumes tokens and builds a tree structure that represents the syntactic organization of the program. It checks that the tokens form a valid program according to the language grammar. It does not check types or semantics: only syntax.

For `let x = 42;`, the parser produces an AST node representing a variable declaration. The tree captures that “x” is being declared and initialized with the integer value 42.

Phase 3: Semantic Analysis Input: AST + symbol tables Output: Semantically annotated AST

Semantic analysis resolves names, checks scopes and builds symbol tables. It determines what each identifier refers to: a variable, a function, a type. It checks that identifiers are declared before use, that variable shadowing follows the language rules and that the structure of the program is semantically valid.

Phase 4: Type Checking Input: Semantically annotated AST Output: Type-annotated AST or type error

The type checker verifies that types are used correctly. It infers types where annotations are omitted, checks that operations are applied to compatible types and reports type errors when the program is ill-typed. After type checking, every expression and variable in the AST has an associated type.

Phase 5: IR Generation Input: Type-annotated AST Output: Intermediate Representation

The IR generator (or “lowering phase”) transforms the high-level AST into the compiler’s intermediate representation. This is where many language-specific constructs are translated into a uniform representation. For example, a for loop might be lowered to while loop constructs with explicit initialization, condition and increment.

Phase 6: Control Flow Graph Construction Input: IR Output: CFG-structured IR

The control flow graph makes the program’s control flow explicit. Code is organized into basic blocks: sequences of instructions with a single entry point and single exit point. Edges between blocks represent control flow: jumps, branches, returns. The CFG is essential for most optimizations.

Phase 7: SSA Construction Input: CFG-structured IR Output: IR in Static Single Assignment form

SSA form ensures that each variable is assigned exactly once. Where multiple assignments to a variable exist in the source, SSA creates versioned variables (x_1 , x_2 , etc.). Phi functions at control flow merge points select the correct version based on which path was taken. SSA makes data dependencies explicit and dramatically simplifies optimization.

Phase 8: Optimization Input: SSA IR Output: Optimized SSA IR

Optimization passes transform the IR to improve performance while preserving semantics. Dead code elimination removes unreachable code and unused computations. Constant propagation replaces variables with known constant values. Copy propagation eliminates unnecessary variable copies. Loop optimizations move invariant code outside loops, reduce computation strength and eliminate redundant induction variables. These passes run multiple times in a carefully ordered sequence.

Phase 9: Instruction Selection Input: Optimized SSA IR Output: Target-specific instruction sequence

Instruction selection maps abstract IR operations to concrete machine instructions. For x86-64, an IR add instruction might become an `add rax, rbx` instruction. The selector chooses among multiple possible encodings, considering register usage, instruction latency and code size.

Phase 10: Register Allocation Input: Target instruction sequence with virtual registers Output: Instruction sequence with physical registers

The register allocator assigns a limited number of physical CPU registers to the many virtual registers used in the IR. When registers are exhausted, some values must be spilled to memory. The quality of register allocation has a significant impact on runtime performance.

Phase 11: Code Emission Input: Instruction sequence with physical registers Output: Object file

The code emitter generates machine code and wraps it in an object file format (ELF on Linux, Mach-O on macOS, COFF on Windows). The object file contains sections for code, data, symbols, relocations and debugging information. The system linker later combines object files into an executable.

Error Handling as a First-Class Concern

Errors are inevitable. A compiler's users will write buggy code, misspell keywords, use wrong types and invent syntax that does not exist. How the compiler handles these errors defines the user experience more than any other factor.

Good error handling requires:

Precise location information. Every token, AST node and IR instruction should track its source location. When an error occurs, the compiler should report exactly which line and column is problematic. Source spans (ranges from start to end) are more useful than single positions: they allow underlining the problematic code.

Clear messages. Error messages should explain what went wrong in human language, not compiler internals. Instead of "expected type `Int` but found type `Bool` at position 42:3", report "cannot add a boolean to an integer: expected integer but found boolean".

Suggestions. When possible, suggest how to fix the error. If the user wrote “fnction” instead of “function”, suggest the correct spelling. If they used `==` where a language requires `===`, explain the difference.

Error recovery. A compiler that stops at the first error is frustrating. Users want to see all errors in their code so they can fix them in one pass. Good error recovery allows the compiler to continue parsing after an error, skipping enough input to resynchronize and report subsequent errors independently.

The Rust compiler is widely praised for its error messages. When you make a mistake, `rustc` explains what the error means, where it occurred and often how to fix it. This quality is not accidental: it requires deliberate design in the error reporting infrastructure, source location tracking and the error messages themselves.

Lumina adopts similar principles. Every phase tracks source locations and reports errors with spans. The error type is unified across all phases so that errors from lexing, parsing, type checking and code generation all appear in the same format.

Frontend-IR and Backend Separation

The separation between front end and back end is one of the most important architectural decisions in compiler design. The boundary is the intermediate representation: the front end produces IR, the back end consumes IR and the optimizer sits in between.

This separation enables language independence. You can add support for a new language by writing a front end that produces the compiler’s IR. You can add support for a new target by writing a back end that consumes the IR. The optimizer needs no changes. This is exactly how LLVM works: Clang provides the C front end, `rustc` provides the Rust front end and the LLVM optimizer and back end serve both.

The quality of the IR is critical. A poor IR either loses important information needed for optimization or encodes too much target-specific detail that limits portability. LLVM IR strikes a balance: it is low-level enough that optimizations can reason about performance, but high-level enough that it is portable across architectures.

Lumina uses a custom IR for the first several phases of the back end, giving us full control over the optimization pipeline. Later, we will also show how to

integrate with LLVM, demonstrating both approaches: building your own IR versus reusing an existing framework.

The Lumina Build System and Project Layout

A production compiler is a large software project that requires careful organization. Lumina uses Rust's Cargo build system with a workspace containing multiple crates. The workspace layout is:

```
1 lumina/
2   Cargo.toml           (workspace definition)
3   lumina-lexer/        (lexical analysis)
4     src/
5       lib.rs
6       token.rs
7       lexer.rs
8       error.rs
9   lumina-parser/       (parsing)
10    src/
11      lib.rs
12      ast.rs
13      parser.rs
14      error.rs
15  lumina-semantic/      (name resolution, symbol tables)
16    src/
17      lib.rs
18      symbols.rs
19      resolve.rs
20      error.rs
21  lumina-type/          (type checking)
22    src/
23      lib.rs
24      types.rs
25      checker.rs
26      error.rs
27  lumina-ir/            (intermediate representation)
28    src/
29      lib.rs
30      ir.rs
31      cfg.rs
32      ssa.rs
33      lowering.rs
34  lumina-opt/           (optimization passes)
35    src/
36      lib.rs
37      dce.rs
```

```
38     cprop.rs
39     loops.rs
40 lumina-backend/      (code generation)
41     src/
42     lib.rs
43     instruction.rs
44     regalloc.rs
45     emit.rs
46     abi.rs
47 lumina-driver/      (main executable)
48     src/
49     main.rs
50     args.rs
51     pipeline.rs
52 tests/              (end-to-end tests)
53     basic.lum        (test files)
```

Each crate has a single responsibility and exposes a clean public API. The driver orchestrates the pipeline: it reads command-line arguments, invokes each phase in sequence and reports errors or produces output.

This modular layout has several benefits:

- Independent testing: Each crate can be tested in isolation with unit tests focused on its specific functionality.
- Incremental compilation: Cargo recompiles only the crates that changed, speeding up development.
- Code reuse: Other tools can depend on individual crates without pulling in the entire compiler. For example, a language server could depend on `lumina-parser` and `lumina-type` without needing the backend.
- Clear boundaries: Well-defined crate boundaries prevent tight coupling between phases.

The build system also supports different build configurations. Debug builds enable extra logging and assertions. Release builds optimize the compiler itself for performance. Testing builds include test infrastructure for fuzzing and differential testing.

Design Trade-offs in Compiler Architecture

Every compiler architecture involves trade-offs. Understanding these trade-offs helps you make informed decisions for your own compiler.

Monolithic vs modular. A monolithic compiler is simpler to start but harder to maintain. A modular compiler has more upfront design cost but scales better. LLVM chose modularity from the beginning, which enabled its growth into the dominant compiler framework. GCC started as a single compiler and evolved into a modular architecture over decades.

Custom IR vs existing framework. Building your own IR gives you full control and avoids framework overhead, but you must implement all optimizations and targets yourself. Using LLVM gives you battle-tested optimization and dozens of targets, but adds complexity, binary size and coupling to LLVM's design decisions. Most real-world compilers make a pragmatic choice: use LLVM for the backend while building a custom front end and IR for language-specific needs.

Hand-written vs generated parsers. Hand-written parsers give better error messages, more control over parsing behavior and often better performance. Generated parsers provide grammar validation and require less initial implementation work. Modern compilers increasingly use hand-written parsers: Clang, GCC (since version 4.1), Go (since version 1.6) and rustc all use hand-written recursive descent parsers.

SSA vs non-SSA IR. SSA form dramatically simplifies optimization but adds complexity to IR construction and maintenance. Almost all modern optimizing compilers use SSA form: LLVM, GCC, V8, HotSpot, Go. Non-SSA IR is simpler but limits optimization capabilities.

Single-pass vs multi-pass compilation. A single-pass compiler compiles in one sweep through the source, which is fast but limits optimization. A multi-pass compiler makes multiple passes over the IR, enabling more sophisticated optimization. Production compilers are multi-pass.

Summary

Compiler architecture is about organizing complexity. A production compiler transforms programs through multiple phases, each with a well-defined responsibility and a well-defined interface. The modular architecture separates concerns so that changes in one area do not ripple through the entire system.

Lumina follows proven architectural patterns from production compilers. It uses a multi-phase pipeline with clear separation between front end, middle

end and back end. Each phase operates on a specific representation of the program and communicates with other phases through well-defined interfaces. Error handling is designed as a first-class concern, not an afterthought. The build system supports independent testing and incremental development.

In the next chapter, we will discuss language design considerations and define the Lumina language specification. Language design decisions directly affect compiler complexity, so we will examine how to design a language that is both pleasant to use and efficient to compile.

Chapter 3: Language Design for Implementation

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

Language Design and Compiler Complexity

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

Syntax Design Choices

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

Type System Implications

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

Memory Model and Ownership

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

Evaluation Strategy and Semantics

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

Standard Library and Runtime Expectations

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

The Lumina Language Specification

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

Implementation Language: Why Rust

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

Summary

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

Chapter 4: Lexical Analysis

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

The Role of the Lexer

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

Token Types and Lexical Rules

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

Building a Hand-Written Lexer

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

Understanding the Lexer Code

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

Error Handling in Lexical Analysis

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

Lexer vs. Parser Responsibilities

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

Lexer Engineering and Whitespace Handling

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

Testing the Lexer

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

Summary

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

Chapter 5: Parsing Fundamentals

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

From Tokens to Structure

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

Context-Free Grammars and Syntax

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

LL vs LR Parsing

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

Recursive Descent Parsing

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

Parser Combinators vs Hand-Written

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

Error Recovery in Parsing

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

Implementing Lumina Parse with Recursive Descent

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

Understanding the Parser Code

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

Summary

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

Chapter 6: Abstract Syntax Trees

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

Concrete Syntax Trees vs Abstract Syntax Trees

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

AST Node Design and Representation

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

Type Safety and Pattern Matching

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

Source Location Tracking

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

Desugaring and Pre-Transformation

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

Implementing Lumina AST

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

AST Traversal Patterns

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

Summary

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

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/moderncompilerengineeringfromsourcecodetomachinecode>

What Semantic Analysis Does

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

Symbol Tables and Scope

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

Name Resolution Algorithms

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

Declarations and Definitions

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

Shadowing and Visibility Rules

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

Semantic Errors and Diagnostics

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

Implementing Lumina Semantic Analysis

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

Understanding the Semantic Analyzer

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

Summary

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

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/moderncompilerengineeringfromsourcecodetomachinecode>

Static vs Dynamic Typing

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

Type Checking Algorithms

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

Type Inference vs Type Annotation

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

Type Errors and Suggestions

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

Implementing Lumina Type Checker

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

Summary

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

Chapter 9: Intermediate Representations

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

Why Intermediate Representations?

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

Three-Address Code and SSA

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

High-Level vs Low-Level IRs

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

IR Design Principles

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

Implementing Lumina IR

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

Lowering AST to IR

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

Summary

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

Chapter 10: Control Flow Graphs

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

Basic Blocks and Control Flow

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

Building Control Flow Graphs

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

Dominators and Post-Dominators

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

Natural Loops and Loop Headers

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

CFG Optimization

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

Implementing Lumina CFG

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

Summary

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

Chapter 11: Static Single Assignment

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

SSA and Its Benefits

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

Phi Functions and Versioning

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

SSA Construction Algorithms

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

Sparse Conditional Constant Propagation

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

Optimizing SSA Form

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

Implementing Lumina SSA

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

Summary

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

Chapter 12: Data Flow Analysis

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

Data Flow Analysis Fundamentals

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

Reaching Definitions and Available Expressions

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

Live Variable Analysis

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

Interprocedural Analysis

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

Analysis Precision and Performance

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

Implementing Lumina Data Flow

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

Summary

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

Chapter 13: Core Optimizations

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

The Optimization Framework

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

Dead Code Elimination

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

Constant Propagation and Folding

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

Copy Propagation

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

Common Subexpression Elimination

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

Summary

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

Chapter 14: Loop Optimizations

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

Loop Optimization Opportunities

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

Loop-Invariant Code Motion

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

Strength Reduction

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

Induction Variable Elimination

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

Loop Unrolling and Peeling

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

Implementing Lumina Loop Optimizations

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

Summary

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

Chapter 15: Advanced Optimizations

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

Function Inlining Strategies

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

Escape Analysis

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

Tail Call Optimization

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

Algebraic Simplification

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

Profile-Guided Optimization

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

Implementing Lumina Inliner

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

Summary

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

Chapter 16: Instruction Selection

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

The Instruction Selection Problem

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

Tree-Pattern Matching

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

Instruction Scheduling

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

Peephole Optimization

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

Lowering for x86-64

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

Implementing Lumina Instruction Selection

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

Summary

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

Chapter 17: Register Allocation

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

Register Allocation Problems

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

Graph Coloring Algorithms

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

Linear Scan Allocation

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

Spilling Strategies

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

Register Pressure

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

Implementing Lumina Register Allocator

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

Summary

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

Chapter 18: Code Generation and Assembly

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

Assembly Language and Encoding

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

Machine Code Generation

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

Relocations and Symbols

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

ABI and Calling Conventions

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

Emitting ELF Object Files

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

Implementing Lumina Code Generator

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

Summary

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

Chapter 19: Linking and Loading

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

The Linker's Job

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

Static vs Dynamic Linking

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

Symbol Resolution and Relocation

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

Link-Time Optimization

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

The Loader and Dynamic Linking

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

Integrating Lumina with the Toolchain

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

Summary

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

Chapter 20: Runtime Support

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

What Compilers Need from Runtimes

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

Initialization and Finalization

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

Standard Library Design

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

Error Handling and Panics

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

Memory Management Patterns

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

Building Lumina Runtime

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

Summary

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

Chapter 21: Diagnostics and Debugging

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

Error Message Design

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

Source Maps and Location

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

Diagnostic Suggestions

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

Debug Information: DWARF

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

Debugging the Compiler

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

Implementing Lumina Diagnostics

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

Summary

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

Chapter 22: JIT Compilation

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

When and Why JIT?

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

Compilation at Runtime

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

Tiered Compilation

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

Deoptimization

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

LLVM ORC and Modern JIT Infrastructure

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

JIT Design for Lumina

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

Summary

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

Chapter 23: Incremental and Parallel Compilation

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

The Incremental Compilation Problem

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

Incremental Dependency Tracking

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

Query-Based Incremental Systems

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

Parallel Compilation

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

Distributed Compilation

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

Making Lumina Fast

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

Summary

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

Chapter 24: Compiler Testing and Verification

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

Compiler Testing Strategies

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

Unit Testing Compiler Components

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

End-to-End and Regression Tests

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

Compiler Fuzzing

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

Formal Verification of Compilers

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

Testing Lumina

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

Summary

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

Chapter 25: Performance Engineering

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

Profiling Compiler Performance

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

Compiler Memory Usage

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

Compilation Time Optimizations

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

Cache Efficiency in Compilers

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

Benchmarking the Compiler

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

Profiling Lumina Compilation

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

Summary

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

Chapter 26: Compiler Frameworks and Ecosystems

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

LLVM Architecture and Use Cases

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

GCC: The Veteran Toolkit

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

MLIR: Multilevel Intermediate Representation

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

Craneflirt for WebAssembly

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

Parser Generators: ANTLR, LALRPOP, Tree-Sitter

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

Choosing Your Compiler Stack

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

Summary

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

Chapter 27: Hardware Awareness

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

CPU Architectures and ISAs

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

Instruction Latency and Throughput

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

Memory Hierarchy Awareness

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

Branch Prediction and Control Flow

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

SIMD and Vectorization

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

Hardware Features in Lumina

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

Summary

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

Chapter 28: The Complete Lumina Compiler

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

Architecture Overview

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

Complete Project Structure

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

The Compilation Driver

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

Complete Lumina Program

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

Extending Lumina

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

Summary

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

Chapter 29: Extending and Evolving a Compiler

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

Adding Language Features

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

Supporting New Platforms

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

Scaling the Codebase

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

Maintaining Compiler Quality

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

Compiler Versioning and Breaking Changes

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

Community and Ecosystem

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

Summary

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

Conclusion

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

References

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