MASTERING AST-GREP

A Comprehensive Guide to Structural Code Search, Linting, and Transformation

Herrington Darkholme

Preface

Who Should Read This Book

This book is written for developers who understand programming and wish to master structural code search, linting, and transformation. Readers are expected to possess working knowledge of at least one programming language—the ability to read code, understand basic syntax, and recognize common patterns like function calls, variable declarations, and control flow constructs.

While examples throughout the book draw primarily from JavaScript and TypeScript—languages whose syntax is widely familiar and whose ubiquity in web development makes them natural choices for illustration—the principles and techniques apply universally. ast-grep is a polyglot tool, and this book reflects that breadth. Examples and case studies incorporate C, Go, Python, Rust, and other languages to demonstrate how structural search transcends language boundaries. Readers need not be expert in all these languages; the examples are designed to be comprehensible to anyone with general programming literacy.

The book is not an introduction to programming. It assumes readers have written and debugged code, understand what a function or class is, and have encountered the frustrations of searching through codebases—whether with `grep`, an IDE's search function, or manual inspection. If you have ever needed to find all invocations of a function, identify deprecated API usage, or refactor patterns across multiple files, this book addresses those tasks with precision and automation.

What This Book Covers

This is a comprehensive technical tutorial that progresses from foundational concepts through advanced applications, building expertise systematically. The structure reflects a progression from understanding principles to applying sophisticated techniques:

Introduction (Chapters 1–3): Establishes the conceptual foundation. Why text-based search fails for code. What Abstract Syntax Trees are and how parsers construct them. How tree-sitter enables polyglot parsing. Why structural search represents the correct abstraction for code manipulation. Readers new to these concepts will find this section essential; those already familiar may skim or skip to the practical chapters.

Basic (Chapters 1–8): Introduces pattern-based searching and rule authoring. These chapters teach the core syntax: simple patterns with metavariables, matching multiple nodes with `$$$`, writing rules in YAML, matching by node type with `kind`, applying regular expressions, composing patterns with logical operators (`and`, `any`, `not`), and performing basic rewrites with `fix`. By the end of this section, readers can write practical patterns to search and refactor codebases.

Intermediate (Chapters 1–10): Deepens pattern authorship and introduces tooling integration. Topics include refining ambiguous patterns with context, controlling relational rule scope with `stopBy` and `field`, creating reusable utility rules, constraining metavariable matches, configuring projects with `sgconfig.yml`, testing rules systematically, integrating with editors via language servers, and debugging rule behavior using the playground. This section transforms readers from users who can write patterns into practitioners who can build robust, tested rule systems.

Advanced (Chapters 1–12): Explores the full depth of ast-grep's capabilities. Advanced pattern syntax interpretation, matching strictness levels (`cst`, `smart`, `ast`, `relaxed`, `signature`), recursive utility rules, metavariable transformation (`replace`, `substring`, `convert`), experimental rewriters, custom match labels for enhanced diagnostics, programmatic API usage, command-line integration modes (JSON output, stdin input), custom language registration, language injection for embedded code (CSS in JavaScript, GraphQL in templates), and the tool's inherent limitations. These chapters enable library authors, tooling developers, and power users to leverage ast-grep's full potential.

How to Use This Book

Read sequentially if you are new to structural code search. The progression builds mental models deliberately: understanding why AST-based search matters precedes learning pattern syntax, which precedes composing complex rules. Each concept serves as foundation for subsequent material.

Use as a reference if you have specific goals. The chapter titles indicate topics clearly. Searching for how to match multiple arguments? Chapter on `$$$`. Need to constrain a metavariable? Intermediate chapter on constraints. Want to register a custom language? Advanced chapter on custom language support.

Experiment while reading. ast-grep provides a playground for interactive pattern development. The command-line tool responds instantly. Testing hypotheses—"Does this pattern match that code?"—requires seconds. The book presents numerous examples; execute them, modify them, observe how changes affect behavior. This active engagement accelerates learning far beyond passive reading.

Expect precision. This book does not simplify for brevity. Technical accuracy takes precedence. When a concept has subtle nuances—named versus unnamed nodes, CST versus AST, pattern strictness levels—those nuances are explained thoroughly. The goal is not merely to teach syntax but to build accurate mental models that enable readers to predict tool behavior and reason about edge cases independently.

Conventions

Code examples are syntax-highlighted and labeled by language. Patterns appear as they would be written in the command line or YAML files. Tree structures are rendered in indented text format with node types and relationships made explicit.

Terminology is consistent throughout. The term "metavariable" always refers to pattern variables like `$VAR` that capture subtrees. "Node" refers to elements in the syntax tree. "Pattern" refers to code fragments used for structural matching. "Rule" refers to YAML specifications combining patterns with metadata like severity and messages.

Cross-references appear as natural text: "as described in the pattern syntax chapter" or "see the section on relational rules." The table of contents provides direct navigation to specific topics.

Acknowledgments

ast-grep exists because tree-sitter exists. The parser generator infrastructure that tree-sitter provides—fast, incremental, error-tolerant parsers for dozens of languages—makes polyglot structural search practical. This book references tree-sitter's design decisions frequently because understanding the parsing layer clarifies ast-grep's capabilities and constraints.

The examples and case studies throughout this book draw from real-world code patterns encountered in production codebases. The frustrations that motivate structural search—brittle regular expressions, false positives from text search, verbose AST manipulation code—reflect the collective experience of the development community.

Final Note

Structural code search is not a novelty; it is the correct abstraction for code analysis and transformation tasks. Text is too low-level. Compiler APIs are too complex. Pattern-based structural matching operates at the right level: syntactic structures expressed in the language's own syntax.

Mastering ast-grep means mastering this abstraction. The investment yields returns across your career—every codebase you encounter, every refactoring task you face, every linting rule you author. The tool becomes an extension of your capability to understand and manipulate code at scale.

This book provides the foundation for that mastery.

Recommendation

Structured Thinking—The Evolutionary Weapon for Developers in the AI Era

In the world of high-performance toolchains and cross-language ecosystems, I have always adhered to one principle: **An exceptional tool should not merely solve a problem; it should redefine the boundaries of efficiency.**

As a developer who frequently moves between TypeScript and Rust, I understand the immense value that high-performance, zero-dependency underlying tools bring to the entire ecosystem. When I first encountered **ast-grep**, I was fully committed to building **napi-rs**, working to eliminate the long-standing obstacles Node.js developers faced when invoking native Rust capabilities. As an early sponsor of the project, I was deeply moved by the vision presented by Herrington Darkholme — he wasn't just writing a faster search tool; he was crafting a "scalpel" capable of understanding code semantics.

Technical Synergy and Ecosystem Convergence

Architecturally, ast-grep shares a deep connection with my work. It utilizes **napi-rs** as the cornerstone for its cross-language integration, allowing it to maintain the extreme performance of Rust while seamlessly integrating into the familiar ecosystem used by front-end developers. This dual pursuit of low-level performance and ultimate ease of use perfectly aligns with the "Node-API revolution" I have advocated for years.

This technical trust has blossomed further in the AI era. In my recently initiated project, **mlx-node**—a framework bringing GPU-accelerated deep learning to Node.js—we are utilizing its powerful inference and training capabilities to perform targeted **Fine-tuning** on models. Our goal is clear: to have models deeply study and master the syntax and logic of ast-grep, enabling them to generate precise ast-grep rules more proficiently and efficiently. This attempt to combine high-performance frameworks with structural tools represents an exploration into the next frontier of code governance.

A "Dual Multiplier" in the AI Wave

As we stand at this juncture in 2026, where AI is profoundly reshaping development paradigms, the value of ast-grep is undergoing a qualitative shift from a "manual tool" to an "intelligent foundation":

- **In the Pre-AI Era**: It served as the "icebreaker" for large-scale code governance. Whether accurately capturing fragile anti-patterns across millions of lines of code or performing cross-framework architectural migrations, ast-grep freed us from the blindness of traditional text-based search (grep) and enabled precise refactoring based on semantic structure.

- **In the AI Era**: It acts as both an "amplifier" of AI efficiency and a "sanitizer" of its output. While generative AI can rapidly produce code, its output often carries a degree of randomness. By fine-tuning models to master ast-grep, we can enable AI to automatically generate rigorous validation rules. These rules serve as "guardrails" to standardize AI-generated output, ensuring it meets the rigors of engineering standards. Furthermore, the precise code snippets extracted by ast-grep provide higher-quality context for models, moving AI-assisted programming from "probabilistic guessing" toward "logical deduction."

Why Read This Book?

Herrington is not only the creator of ast-grep but also a rare expert in the open-source community who can balance low-level "black magic" with high-level engineering practice.

This book systematically outlines the progression from AST basics to complex refactoring rules. Whether you are a developer looking to improve toolchain performance with Rust or an architect exploring how to empower large-scale code governance with AI, this book will provide you with a complete model for "structured thinking."

At Void Zero, we are dedicated to building the next generation of AI-integrated developer toolchains. I firmly believe that **ast-grep is an indispensable semantic engine within that future toolchain.** I hope every reader finds in this book the "legendary sword" needed to navigate the vast oceans of large-scale code with ease.

Broooooklyn *Member of Void Zero / Creator of napi-rs / Initiator of mlx-node*

Table of Contents

Preface 3

Introduction

1. Introduction to ast-grep 13
2. Abstract Syntax Trees, Parsers, and the Transformation of Source Code 20
3. Getting Started with ast-grep 28

Basic

1. Pattern-Based Matching 35
2. Metavariable Substitution 41
3. Rule-Based Pattern Specification 47
4. Node Type Matching with `kind` 56
5. Text Content Matching with `regex` 67
6. Relational Rules 68
7. Composite Rules 69
8. Structural Code Transformation 102

Intermediate

1. Pattern Objects with Context 115

2. Relational Rule Refinement: stopBy and field 125

3. Utility Rules: Reusable Matching Logic 137

4. Constraining Meta-Variable Matches 151

5. Project Configuration with `sgconfig.yml` 161

6. Testing Your Rules: Building Confidence with `ast-grep test` 173

7. Editor Integration: Real-Time Code Analysis with VSCode and LSP 184

8. The ast-grep Playground: A Browser-Based Development Laboratory 199

9. Debugging Rules: Systematic Diagnosis and Resolution 208

10. Leveraging AI Assistants for Rule Development 218

Advanced

1. Deep Dive: How ast-grep Understands Your Patterns 228

2. Refining Match Precision: Strictness Semantics 239

3. Advanced Fixes: Expanding the Rewrite Range 251

4. Transforming Captured Values: String Operations on Metavariables 259

5. Rewriters: Compositional Code Transformation 273

6. Customizing Match Highlighting with Labels 289

7. Programmatic Access: The ast-grep API 297

8. Pipeline Integration: JSON Output and Stdin Mode 323

9. Extending ast-grep with Custom Languages 345

10. Handling Embedded Code: Language Injection 361

11. Architectural Boundaries and Tool Selection 376

Introduction

Introduction to ast-grep

What is ast-grep?

ast-grep is a structural code search, linting, and rewriting tool that operates on Abstract Syntax Trees rather than raw text. It provides the simplicity and speed of command-line tools like grep while delivering the precision and syntax awareness of compiler-based analysis tools. Where text-based search treats code as strings and regular expressions struggle with nested structures, ast-grep understands code as the compiler does—as a hierarchical tree of syntactic elements.

ast-grep fundamentally differs from text-based tools by operating on a structured, tree-like representation of your code. This distinction enables searches that are immune to formatting variations, comprehend nested structures, and can distinguish between textually identical but structurally different code elements. A search for a function call will match that function call regardless of argument formatting, whitespace, or line breaks—matching the structural concept rather than a textual pattern.

Consider this invocation:

```
ast-grep --pattern 'var code = $PAT' --rewrite 'let code = $PAT' --lang js
```

This command transforms every `var` declaration into a `let` declaration across JavaScript files. The metavariable `$PAT` captures arbitrary expressions, enabling the rewrite to preserve the assigned value regardless of its complexity. The pattern matches the syntactic structure, not a text string; formatting and whitespace variations have no effect on the match.

The Limitations of Text-Based Code Search

Text-based search tools like grep operate on raw character sequences. While fast and universally applicable, this approach fails when confronted with the structural nature of code. Regular expressions, despite their power for text processing, cannot reliably parse nested or recursive structures—a fundamental characteristic of nearly all programming languages.

When Text Search Breaks Down

Consider searching for a function call: `console.log(message)`. A naive grep or regex approach encounters immediate problems:

Formatting and structural variations: Function calls exhibit numerous syntactic variations that confound text-based matching. Whitespace, line breaks, trailing commas, and argument formatting all affect text representation while preserving structural identity:

```
console.log(message)
console.log(
  message
)
console.log( message )
console . log ( message )
console.log(
  message,
)
console.log(
  message1,
  message2,
  message3
)
```

A text pattern `console.log(message)` matches only the first variant. Regular expressions can accommodate whitespace variations with `\s*` and optional tokens, but such patterns become unwieldy: `console\s*\.\s*log\s*\(\s*message\s*,?\s*\)`. This pattern still fails when line breaks separate tokens or when additional arguments appear. Accounting for all formatting permutations produces brittle, unmaintainable expressions.

Nested structures: Code's recursive nature fundamentally exceeds regular expression capabilities. Function arguments may themselves contain function calls, creating arbitrary nesting:

```
console.log(format(getMessage(user.name, user.id)))
```

Matching this structure with regex requires patterns that account for nested parentheses—a context-free language construct that regular expressions, being regular languages, cannot properly parse. Consider attempting to extract the argument from `console.log`:

```
console\.log\(([^()]*(?:\([^()]*\))?[^()]*)\)
```

This pattern attempts to match one level of nesting through non-capturing groups and optional nested parentheses. It fails immediately with two levels of nesting like `format(getMessage(user.name, user.id))`. Extending the pattern to handle arbitrary nesting depth is theoretically impossible with pure regular expressions—the language class lacks the expressiveness. Practical regex engines offer extensions (recursive patterns, backreferences), but these sacrifice performance and produce incomprehensible expressions.

Cognitive overhead: Beyond theoretical limitations, regular expressions impose substantial maintenance burden. Patterns that handle realistic code complexity become cryptic character sequences requiring careful analysis to understand. Consider extracting a function name and its first argument from any function call:

```
([a-zA-Z_$][a-zA-Z0-9_$]*)\s*\.\s*([a-zA-Z_$][a-zA-Z0-9_$]*)\s*\(\s*([^,)]+)
```

This pattern matches `console.log(message)` with three captured groups: the object identifier, the method identifier, and the first argument. Understanding what it captures requires parsing the character class definitions, quantifiers, and escaped metacharacters. Modifying it to handle additional cases—optional chaining, computed properties, template literals—compounds the complexity. Six months later, the author cannot explain their own pattern without careful study.

False matches in strings and comments: Text search cannot distinguish between code and non-code:

```
// Don't call console.log() in production
const warning = "Remember: console.log(data) is slow";
console.log(data);
```

A grep for `console.log` matches all three lines, despite only the third being an actual function invocation.

Syntax structure: Consider finding a variable assignment. The text `x = 5` could appear in multiple contexts:

```
let x = 5;            // Variable declaration with initialization
x = 5;                // Assignment expression
class A { x = 5 }     // Field initialization
const msg = "x = 5"; // String literal
```

Regular expressions operate on character sequences and cannot distinguish these syntactic contexts. They lack awareness of scope, syntax, and program structure.

Why AST-Based Search Succeeds

Abstract Syntax Trees represent code as hierarchical structures where formatting is irrelevant and syntactic meaning is explicit. Each node in the tree represents a syntactic construct—a function call, an identifier, a binary operation—with defined relationships to other nodes.

When code is parsed into an AST:

- Whitespace and formatting disappear; they are not part of the tree
- Comments are typically omitted or stored separately
- Syntactic elements are categorized by type: `function_call`, `identifier`, `assignment_expression`
- The tree structure reflects nesting and scope relationships

For the earlier `console.log(message)` example, the AST representation abstracts away formatting:

```
call_expression
├── member_expression
│   ├── object: identifier ("console")
│   └── property: identifier ("log")
└── arguments
    └── identifier ("message")
```

This tree structure is identical regardless of whitespace, line breaks, or comment placement. A pattern matching this structure matches all formatting variants automatically.

How ast-grep Solves the Problem

ast-grep provides a pattern matching system that operates on Abstract Syntax Trees while maintaining the simplicity of command-line text search. Rather than requiring developers to write verbose AST traversal code or construct complex regular expressions, ast-grep enables patterns written in the target language itself.

Pattern-Based Matching

The core mechanism: use code to search code. A pattern is written in the syntax of the target language, and ast-grep matches that syntactic structure in the codebase. Metavariables—identifiers beginning with `$`—act as wildcards that match and capture subtrees.

The pattern `console.log($MSG)` matches any invocation of `console.log` with a single argument, capturing that argument in the metavariable `$MSG`. This pattern matches:

- `console.log(message)` — captures `message`
- `console.log("Hello")` — captures `"Hello"`
- `console.log(x + y)` — captures `x + y`
- `console.log(fn())` — captures `fn()`

The pattern ignores formatting differences. These all match:

```
console.log(message)
console.log( message )
console.log(
  message
)
```

Language Support via Tree-Sitter

ast-grep leverages tree-sitter, a parser generator library that provides fast, incremental parsers for numerous languages. Tree-sitter generates parsers from grammar specifications, producing Concrete Syntax Trees—detailed tree representations that include all syntactic elements.

ast-grep uses tree-sitter to parse both the codebase and the search patterns, enabling multiple language support: the same pattern-matching principles apply across languages. The tool currently supports:

Language Domain	Languages
System Programming	C, C++, Rust
Server-Side Programming	Go, Java, Python, C#
Web Development	JavaScript, JSX, TypeScript, TSX, HTML, CSS
Mobile Development	Kotlin, Swift
Configuration	JSON, YAML, HCL
Scripting and Etc	Lua, Nix

Each language requires a tree-sitter grammar, which defines the parsing rules and node types for that language. ast-grep's pattern matching operates on the resulting AST regardless of the source language.

Core Capabilities

ast-grep provides three primary modes of operation, each building on AST-based pattern matching:

Search: The command-line tool performs structural search across codebases, scanning thousands of files in seconds. Patterns match structural code structures, eliminating false positives from comments, strings, and formatting variations.

Lint: The rule system enables custom linting rules defined in YAML. Rules compose patterns with logical operators, allowing arbitrarily complex matching predicates. Each rule specifies a pattern, a severity level, and a message, integrating with development workflows as a linter or language server.

Rewrite: Patterns can be paired with rewrite templates, enabling automated code transformations. Metavariables captured during matching are substituted into the rewrite template, allowing refactoring operations that preserve captured expressions. The programmatic API provides fine-grained control for complex transformations or code generation.

Why Structural Search Matters

Structural search operates at the correct level of abstraction for code. Text is too low-level; it conflates syntactic structure with formatting. AST manipulation APIs are too low-level in a different way; they expose implementation details and require verbose traversal logic.

Structural search with pattern matching provides extra precision—patterns match code structures, not character sequences—while maintaining simplicity—patterns are written in the target language, not as API calls or complex regular expressions.

This approach enables:

Precise matching: Patterns match code constructs while ignoring formatting. Searches for a function call match only actual invocations, not textual occurrences in strings or comments.

Automatic handling of variations: Formatting differences, line breaks, trailing commas, and whitespace variations do not affect pattern matching. The AST representation normalizes these variations.

Metavariable capture: Captured subtrees can be reused in rewrite templates, enabling refactoring operations that preserve complex expressions. A pattern can capture a function's arguments and rearrange them, or extract a variable's initializer and relocate it.

Syntax awareness: Patterns distinguish between syntactic contexts. A pattern for an assignment expression does not match equality comparisons or variable declarations with different meaning, even when the text appears similar.

By combining the accessibility of grep with the precision of parser-based tools, ast-grep makes structural search practical for everyday development workflows.

Comparison with AST Manipulation APIs

Direct AST manipulation through compiler APIs provides precision but imposes significant cognitive overhead. Consider matching `console.log` in JavaScript using Babel's AST API:

```
path.parentPath.isMemberExpression() &&
path.parentPath.get('object').isIdentifier({ name: 'console' }) &&
path.parentPath.get('property').isIdentifier({ name: 'log' })
```

This code requires understanding Babel's node traversal API, the specific methods for type checking and property access, and the structure of member expressions. The equivalent ast-grep pattern:

```
ast-grep -p "console.log"
```

The pattern is the code itself. No API methods, no traversal logic, no node type predicates. ast-grep parses the pattern into an AST and matches that structure against the target code.

This distinction extends beyond simplicity. AST manipulation APIs are language-specific; Babel handles JavaScript, while Python requires `ast` module knowledge, and Go requires `go/ast` familiarity. Each API has distinct conventions, methods, and node types. ast-grep's pattern syntax remains consistent across languages—the pattern is always written in the target language's own syntax.

As Jobert Abma, co-founder of HackerOne, observed:

> *The internal AST query interfaces those tools offer are often poorly documented and difficult to write, understand, and maintain.*

ast-grep addresses this by eliminating the need to learn AST manipulation APIs. The pattern language is the programming language itself.

Design Characteristics

Several design decisions distinguish ast-grep from alternative code tools:

Performance: Implemented in Rust with parallel processing, ast-grep handles tens of thousands of files in seconds. The tree-sitter parsers are fast and incremental, enabling responsive interactive use.

Progressive adoption: The tool supports a progression from simple command-line invocations to sophisticated rule systems to programmatic API usage. A developer can begin with a single pattern string, advance to YAML rule definitions combining multiple patterns, and ultimately leverage the API for complex transformations or custom tooling.

Integrated tooling: The command-line tool includes a test framework for rule authors, a language server for editor integration, and interactive modification capabilities. These features are available upon installation without additional configuration.

This combination of simplicity, performance, and tooling integration makes ast-grep practical for routine development tasks while remaining powerful enough for complex code analysis and transformation workflows.