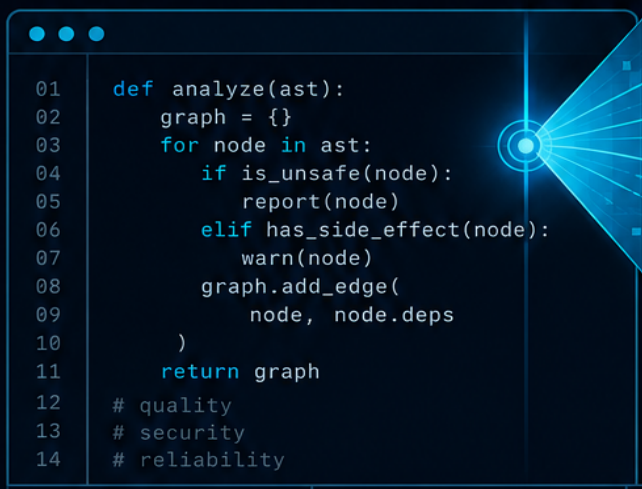
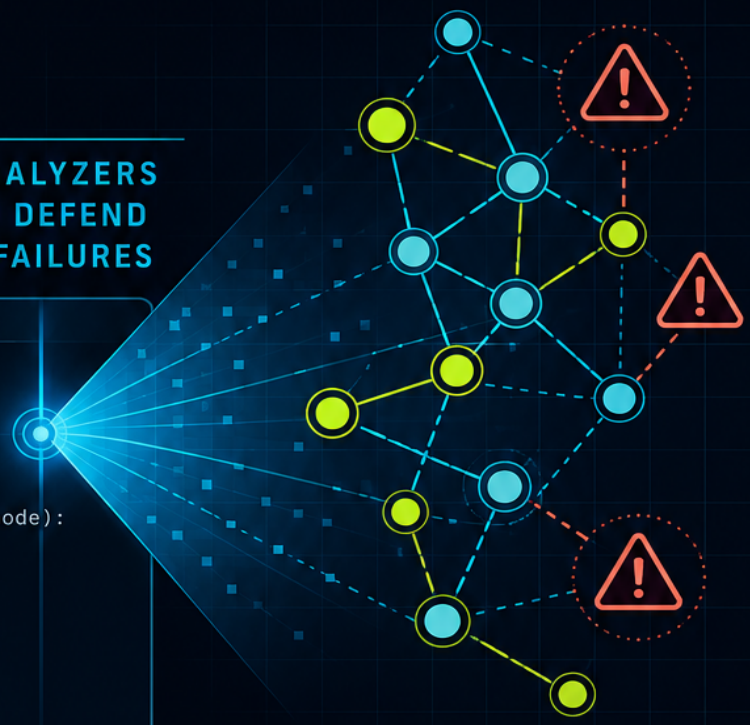
STATIC ANALYSIS FOR AI-GENERATED CODE

BUILDING PRODUCTION ANALYZERS TO DETECT, DIAGNOSE, AND DEFEND AGAINST AI CODE QUALITY FAILURES



```
01  def analyze(ast):
02      graph = {}
03      for node in ast:
04          if is_unsafe(node):
05              report(node)
06          elif has_side_effect(node):
07              warn(node)
08          graph.add_edge(
09              node, node.deps
10      )
11      return graph
12  # quality
13  # security
14  # reliability
```



PHILLIPS MÜLLER

Static Analysis for AI-Generated Code

Building Production Analyzers to Detect, Diagnose, and Defend Against AI Code Quality Failures

Steve Publications

This book is available at
https://leanpub.com/staticanalysisforai-generatedcode

This version was published on 2026-07-27

Contents

Building Production Analyzers to Detect, Diagnose, and Defend Against AI Code Quality Failures **1**

Chapter 1: The Static Analysis Imperative in the AI Era **2**
How AI Code Generation Changed Software Engineering 2
The New Defect Taxonomy: AI-Specific Failure Modes 3
Why Dynamic Testing Is Not Enough 4
Static Analysis as a Force Multiplier 5
The Cost of Trusting Generated Code 5
What This Book Will Teach You 6

Chapter 2: Compiler Fundamentals for the Static Analyst **8**
The Compiler Pipeline: From Source to Object 8
Lexical Analysis and Tokenization 8
Parsing and Grammar Design 15
Error Recovery in Real Compilers 23
Why You Need This Background as an Analyst 24

Chapter 3: Abstract Syntax Trees and Code Representation **26**
What an AST Is (and Is Not) 26
Building an AST from Tokens 26
The Visitor Pattern for Tree Traversal 26
Implementing a Concrete AST Visitor 26
Tree Transformations and Rewrites 26
Comparing ASTs: Diffing Generated Code 26
ASTs as the Foundation for Analysis 27

Chapter 4: Control Flow Graphs and Program Structure **28**
From AST to Control Flow Graph 28
Basic Blocks and Dominators 28
Reachability Analysis 28

Detecting Dead Code in AI Output 28
Loop Analysis and Termination Questions 28
Exception Flow and Non-Local Control 28
Why CFGs Matter for AI Code Analysis 29

Chapter 5: Data Flow Analysis and Symbol Resolution 30
The Data Flow Problem 30
Forward vs Backward Analysis 30
Reaching Definitions 30
Live Variable Analysis 30
Symbol Tables and Scope Resolution 30
Call Graph Construction 30
Why Data Flow Matters for AI Code Analysis 31

Chapter 6: Type Systems and Type-Based Analysis 32
What Types Are For 32
Static vs Dynamic Typing for Analysis 32
Type Inference Basics 32
Detecting Type Confusion in Generated Code 32
Null Safety and Option Types 32
Generics, Polymorphism, and AI Pitfalls 32
Why Type-Based Analysis Matters for AI Code 33

Chapter 7: Intermediate Representations and Cross-Language Analysis 34
Why We Need Intermediate Representations 34
LLVM IR as a Case Study 34
Three-Address Code and SSA Form 34
Tree-Sitter: Parsing Anything 34
Building a Language-Agnostic Analyzer 34
Cross-Language Call Graphs 34
Why IRs Matter for AI Code Analysis 35

Chapter 8: Taint Analysis and Security Vulnerabilities 36
The Taint Model 36
Sources, Sinks, and Sanitizers 36
Implementing a Taint Tracker 36
Prompt-Induced Vulnerabilities 36
Injection Attacks in Generated Code 36
Insecure Default Patterns from AI 36
Why Taint Analysis Matters for AI Code 37

Chapter 9: Symbolic Execution and Path-Sensitive Analysis **38**
Concrete vs Symbolic Execution 38
Path Conditions and SMT Solvers 38
Building a Basic Symbolic Executor 38
Path Explosion and Mitigation 38
When Symbolic Execution Is Worth It 38
Concolic Testing for AI Code 38
Why Symbolic Execution Matters for AI Code 39

Chapter 10: Abstract Interpretation and Sound Analysis **40**
The Abstract Interpretation Framework 40
Lattices, Join, and Meet 40
Widening and Narrowing 40
Interval Analysis 40
Pointer Analysis 40
Tradeoffs: Precision vs Performance 40
Why Abstract Interpretation Matters for AI Code 41

Chapter 11: Rule Engines and Custom Linting **42**
The Anatomy of a Static Analysis Rule 42
Pattern Matching on ASTs 42
Building a Rule Engine from Scratch 42
Semgrep-Style Structural Search 42
CodeQL Query Language Design 42
Managing Thousands of Rules 42
Why Rule Engines Matter for AI Code 43

Chapter 12: AI-Specific Code Smells and Detection Strategies **44**
Hallucinated APIs and Nonexistent Functions 44
Inconsistent Coding Patterns and Style Drift 44
Dead Code and Redundant Logic 44
Hidden Dependencies and Missing Imports 44
Context Window Artifacts 44
Incomplete Implementations and TODO Sprawl 44
Duplicated Logic Across Files 45
Over-Engineering and Unnecessary Complexity 45
Why Detecting AI Code Smells Matters 45

Chapter 13: Building a Production Static Analyzer **46**
Architectural Decisions for Scale 46

Incremental Analysis and Caching 46
Parallelization Strategies 46
False Positive Management 46
SARIF and Standardized Output 46
Integration with CI/CD Pipelines 46
Why Production Architecture Matters 47

Chapter 14: The Tooling Ecosystem 48
Language-Specific Analyzers 48
Query-Based Tools (CodeQL, Semgrep) 48
IDE Integration Strategies 49
Analyzing Large AI-Assisted Codebases 49
Choosing Your Stack 49
Building on Existing Infrastructure 49

Chapter 15: Defense in Depth – Integrating Static Analysis into AI Workflows 50
Pre-Generation Guardrails 50
Post-Generation Validation Pipelines 50
Developer Feedback Loops 50
Measuring AI Code Quality 50
The Future of Static Analysis and AI 50
A Practical Checklist 50

Conclusion 52

References 53

Building Production Analyzers to Detect, Diagnose, and Defend Against AI Code Quality Failures

AI code generation has fundamentally changed the shape of software defects. Large language models produce code that looks correct but contains systematic failure modes: hallucinated APIs, prompt-induced vulnerabilities, architectural drift, and subtle logic errors that traditional testing cannot catch efficiently. This book teaches you how to build static analysis tools from first principles specifically designed to detect, diagnose, and defend against these AI-specific quality failures. You will learn compiler fundamentals, control flow and data flow analysis, taint tracking, symbolic execution, abstract interpretation, and rule engine design. Every chapter includes production-ready source code, architectural tradeoff discussions, and practical guidance for integrating analyzers into CI/CD pipelines and IDEs. By the end, you will understand not only how static analysis works but why each technique exists, when to use it, and how it applies to modern AI-assisted software development.

Chapter 1: The Static Analysis Imperative in the AI Era

How AI Code Generation Changed Software Engineering

The adoption of AI coding assistants has accelerated software development at a scale that is difficult to overstate. By the end of 2025, analysts estimated that AI could be responsible for producing up to 90 percent of all new code [6]. Stack Overflow's 2024 Developer Survey found that 63 percent of professional developers were already using AI tools in their daily work [13]. GitHub Copilot alone offers a 46 percent code completion rate, though only about 30 percent of suggestions are accepted by developers, meaning nearly two-thirds of AI-generated proposals are discarded before they enter the codebase [9].

This is not just a productivity story. It is a fundamental shift in how software is constructed. When humans write code, defects emerge from specific cognitive patterns: misunderstanding requirements, overlooking edge cases, misremembering APIs, or being lazy about refactoring. These patterns are predictable and, to some degree, manageable through code review, testing, and experience.

AI-generated code introduces a different defect profile. Large language models do not understand code the way humans do. They predict the next token based on statistical patterns learned from billions of lines of training data. This means they can produce syntactically correct, semantically plausible, functionally broken code with unwavering confidence. The model has no concept of your project's architecture, no awareness of your team's conventions, and no understanding of the security threat model for the system it is helping to build.

The consequences are measurable and concerning. A study by CodeRabbit analyzing 470 open-source GitHub pull requests found that AI-co-authored changes contained approximately 1.7 times more issues than human-only contributions, with critical and major defects occurring at proportionally higher rates [7]. The same study found that logic and correctness issues were 75

percent more common in AI-generated code, readability problems spiked more than threefold, and error handling gaps were nearly twice as prevalent.

GitClear's analysis of 211 million lines of changed code between January 2020 and December 2024 revealed another troubling trend: code duplication has grown fourfold since AI assistants became widespread [13]. Lines classified as copy-pasted clones rose from 8.3 percent to 12.3 percent, while refactored code dropped from 25 percent to under 10 percent. This is not accidental. AI models generate fresh implementations for problems that already have solutions in the codebase, because they cannot see the whole project at once. The result is a codebase slowly fragmenting into parallel, slightly divergent implementations of the same functionality.

The New Defect Taxonomy: AI-Specific Failure Modes

To build effective static analysis tools for AI-generated code, we must first understand what kinds of defects to look for. Research has identified several categories of failures that are either unique to or disproportionately common in AI-generated code [6, 20].

Hallucinated APIs represent one of the most insidious problems. An LLM might generate a call to `pandas.DataFrame.merge_on()` because it sounds plausible and resembles real pandas methods like `merge()` and `groupby()`. The code compiles without errors in many dynamically typed languages. It only fails at runtime, potentially deep inside a deployed service. A 2026 study by Khati et al. demonstrated that AST-based analysis can detect these hallucinations by parsing generated code and validating function calls against a dynamically constructed knowledge base built from library introspection [3].

Prompt-induced vulnerabilities are security flaws that arise directly from how prompts are framed. If a developer asks an AI to "quickly parse user input and query the database," the model may generate SQL concatenation instead of parameterized queries, because it is optimizing for brevity over security. Research on Copilot-generated code found vulnerable suggestions across multiple OWASP Top 25 CWE categories, including injection flaws, insecure cryptographic storage, and broken access control [9]. The model is not malicious; it simply lacks the threat modeling context that a human reviewer would bring.

Context window artifacts emerge when the AI's limited context forces it to make assumptions about code it cannot see. When generating code for

a large file or complex module, the model may truncate its understanding of earlier definitions, leading to what researchers call the "donut hole" problem: the model remembers the beginning and end of a prompt but loses the middle [20]. This can result in incomplete implementations, references to undefined variables, or functions that assume parameters they never receive.

Style drift and architectural erosion occur gradually. Each AI-generated snippet may individually follow reasonable conventions, but collectively they introduce subtle inconsistencies: different error handling patterns across modules, divergent naming schemes, parallel utility functions scattered throughout the codebase. Over time, the codebase loses its architectural coherence. Tools like Drift have been built specifically to detect this pattern fragmentation and measure cross-file structural degradation [20].

Incomplete implementations are everywhere in AI-generated code. Models are trained to produce responses that look complete but often skip the hard parts: proper error handling, resource cleanup, edge case coverage, and security validation. A try-catch block might log an error without actually handling it. A file upload handler might validate the MIME type but not scan for malware. The code looks production-ready at a glance but falls apart under real-world conditions [20].

Why Dynamic Testing Is Not Enough

You might wonder whether comprehensive testing can solve these problems. In theory, yes: if every line of AI-generated code is covered by tests, any bug will be caught before deployment. In practice, this approach has fundamental limitations.

First, tests are themselves often AI-generated, introducing the same defect patterns into the validation layer. An AI might generate a function with a logic error and then generate tests that pass against that buggy behavior, creating a self-consistent but incorrect system.

Second, testing is inherently reactive. It validates behavior for specific inputs but cannot exhaustively verify correctness across all possible execution paths. Static analysis, by contrast, reasons about the code structure itself and can detect entire classes of bugs without executing the program.

Third, many AI-specific defects are structural rather than behavioral. Code duplication, architectural drift, hallucinated APIs that happen to match a

shadow dependency, and inconsistent patterns do not necessarily cause test failures but degrade maintainability and increase long-term risk. These problems require structural analysis, not just execution-based validation.

Fourth, the cost of testing scales with code volume. As AI accelerates development velocity, the amount of new code to test grows proportionally. Static analysis provides a scalable first line of defense that can evaluate millions of lines of code in minutes, flagging issues before they reach the testing stage.

Static Analysis as a Force Multiplier

Static analysis is not a replacement for testing, code review, or developer judgment. It is a force multiplier that makes all of those activities more effective. When properly integrated into the development workflow, static analysis tools provide several critical benefits for AI-assisted development:

Early detection: By running analysis on AI-generated code before it is committed, teams can catch hallucinated APIs, security vulnerabilities, and obvious logic errors at the point of generation rather than discovering them in production.

Consistency enforcement: Custom rules can enforce project-specific conventions, architectural patterns, and security requirements that the AI model has no way to know about. This prevents style drift and ensures that generated code integrates smoothly with existing systems.

Scalable review assistance: Static analysis tools can automatically annotate pull requests with findings, giving human reviewers a prioritized list of things to check rather than requiring them to read every line of AI-generated code manually.

Feedback loop acceleration: When static analysis catches an issue in AI-generated code, that feedback can be fed back into the prompt or the AI workflow, teaching the model to avoid similar mistakes in future generations.

The Cost of Trusting Generated Code

The fundamental mistake teams make with AI coding assistants is treating them as oracles rather than probabilistic generators. When developers accept AI suggestions without scrutiny, they introduce defects at a rate that outpaces

their ability to detect and fix them afterward. Google's 2024 DORA report found that a 25 percent increase in AI usage was associated with a 7.2 percent decrease in delivery stability [20]. SonarSource reported that a 90 percent increase in AI adoption correlated with an estimated 9 percent climb in bug rates, a 91 percent increase in code review time, and a 154 percent increase in pull request size [13].

These numbers tell a clear story: AI accelerates output but degrades quality unless counterbalanced by stronger verification practices. Static analysis is the most practical and scalable of those practices. It does not require changing how developers prompt AI models or how teams organize their work. It works on the code itself, providing an objective, automated assessment of quality and safety regardless of how the code was produced.

What This Book Will Teach You

This book is organized to take you from foundational concepts to production deployment. Chapters 2 through 4 build the compiler fundamentals you need: lexical analysis, parsing, abstract syntax trees, and control flow graphs. These are not academic exercises; they are the data structures and algorithms that every serious static analyzer depends on.

Chapters 5 through 10 cover the core analysis techniques: data flow analysis, type systems, intermediate representations, taint tracking, symbolic execution, and abstract interpretation. Each chapter explains not just how the technique works but why it exists, what problems it solves, and where it breaks down.

Chapter 11 focuses on rule engines and custom linting, showing you how to design flexible systems that let domain experts encode knowledge without writing low-level analysis code. Chapter 12 catalogs the specific failure modes of AI-generated code and explains how to detect each one with static analysis techniques.

Chapters 13 and 14 address production concerns: building analyzers that scale to large codebases, managing false positives, integrating with CI/CD pipelines, and choosing the right tools from the existing ecosystem. Chapter 15 synthesizes everything into a practical defense-in-depth strategy for organizations using AI coding assistants at scale.

Every chapter includes complete, runnable source code. The examples are not toy implementations; they are production-oriented designs that demonstrate real engineering tradeoffs. You will learn how to build analyzers that are fast enough to run in CI, precise enough to earn developer trust, and extensible enough to adapt as AI-generated code evolves.

The goal is not just to teach you static analysis. The goal is to give you the tools and understanding to build a defense against the systematic quality failures that AI code generation introduces, so your team can benefit from the productivity gains without paying the long-term cost in technical debt and security risk.

Chapter 2: Compiler Fundamentals for the Static Analyst

The Compiler Pipeline: From Source to Object

A static analyzer is essentially a compiler that stops before generating machine code. Understanding the compiler pipeline is not optional background knowledge; it is the foundation on which every analysis technique rests. When you know how compilers work, you understand why analyzers are structured the way they are, what information is available at each stage, and where the fundamental limitations come from.

The classic compiler pipeline has three major phases:

Front-end: Takes source code as input and produces an intermediate representation. This phase includes lexical analysis (tokenization), parsing (building a syntax tree), and semantic analysis (type checking, symbol resolution).

Middle-end: Optimizes the intermediate representation using language-independent transformations like dead code elimination, constant folding, and loop optimization.

Back-end: Translates the optimized IR into target-specific machine code.

Static analyzers live primarily in the front-end and middle-end. They consume the same representations that compilers build and apply analysis passes instead of optimization or code generation passes. The key insight is that every piece of information a static analyzer needs is produced by one of these phases.

Lexical Analysis and Tokenization

Lexical analysis is the first step in understanding source code. It converts a raw character stream into a sequence of tokens, which are the meaningful atomic units of the language: keywords, identifiers, operators, literals, and punctuation.

Consider this Python code:

```
x = 42 + foo(bar)
```

A lexer transforms this character stream into the following token sequence:

```
IDENTIFIER("x"), EQUALS("="), NUMBER(42), PLUS("+"),
IDENTIFIER("foo"), LPAREN("("), IDENTIFIER("bar"), RPAREN(")")
```

This may seem trivial, but lexical analysis solves several non-obvious problems. It must handle whitespace and comments (which are significant in some languages and ignored in others). It must distinguish between similar character sequences: in C++, `>>` could be two right-shift operators or a nested template close. It must recognize string literals with escape sequences, numeric literals in various bases, and identifiers that may include Unicode characters.

Here is a production-oriented lexer implementation for a small expression language. This demonstrates the core pattern used by real lexers: scanning character by character, recognizing patterns, and emitting tokens with source location information.

```
from enum import Enum, auto
from dataclasses import dataclass
from typing import List, Optional, Iterator


class TokenType(Enum):
    """Token types recognized by the lexer."""
    IDENTIFIER = auto()
    INTEGER = auto()
    STRING = auto()
    PLUS = auto()      # +
    MINUS = auto()     # -
    STAR = auto()      # *
    SLASH = auto()     # /
    LPAREN = auto()    # (
    RPAREN = auto()    # )
    LBRACE = auto()    # {
    RBRACE = auto()    # }
    SEMICOLON = auto() # ;
    COMMA = auto()     # ,
    EQUALS = auto()    # =
    EQEQ = auto()      # ==
```

```
    NEQ = auto()        # !=
    LT = auto()         # <
    GT = auto()         # >
    LTE = auto()        # <=
    GTE = auto()        # >=
    IF = auto()
    ELSE = auto()
    WHILE = auto()
    RETURN = auto()
    EOF = auto()


@dataclass
class Token:
    """A single lexical token with source location."""
    type: TokenType
    value: str
    line: int
    col: int

    def __repr__(self):
        return f"Token({self.type.name}, {self.value!r},
        ↪  {self.line}:{self.col})"


class LexerError(Exception):
    """Raised when the lexer encounters invalid input."""
    def __init__(self, message: str, line: int, col: int):
        super().__init__(f"Line {line}, col {col}: {message}")
        self.line = line
        self.col = col


class Lexer:
    """
    Lexical analyzer for a small expression language.

    This implementation demonstrates production-oriented patterns:
    - Source location tracking for every token (critical for error reporting)
    - Multi-character operator recognition with backtracking
    - String literal handling with escape sequences
    - Comment skipping
    - Clear error messages with positions
    """

    KEYWORDS = {
        "if": TokenType.IF,
        "else": TokenType.ELSE,
        "while": TokenType.WHILE,
        "return": TokenType.RETURN,
```

```
    }

    def __init__(self, source: str):
        self.source = source
        self.pos = 0
        self.line = 1
        self.col = 1

    def _current(self) -> Optional[str]:
        """Return current character or None if at end."""
        if self.pos < len(self.source):
            return self.source[self.pos]
        return None

    def _advance(self) -> str:
        """Consume and return the next character."""
        ch = self.source[self.pos]
        self.pos += 1
        if ch == "\n":
            self.line += 1
            self.col = 1
        else:
            self.col += 1
        return ch

    def _peek(self, offset: int = 1) -> Optional[str]:
        """Look ahead without consuming."""
        idx = self.pos + offset
        if idx < len(self.source):
            return self.source[idx]
        return None

    def _skip_whitespace_and_comments(self):
        """Skip whitespace and single-line comments."""
        while True:
            ch = self._current()
            if ch is None:
                break
            if ch in " \t\r\n":
                self._advance()
            elif ch == "/" and self._peek() == "/":
                # Single-line comment: skip to end of line
                while self._current() is not None and self._current() != "\n":
                    self._advance()
            else:
                break

    def _read_identifier(self) -> Token:
        """Read an identifier or keyword."""
        start_line, start_col = self.line, self.col
```

```
        result = []
        while True:
            ch = self._current()
            if ch is not None and (ch.isalnum() or ch == "_"):
                result.append(self._advance())
            else:
                break
        text = "".join(result)
        token_type = self.KEYWORDS.get(text, TokenType.IDENTIFIER)
        return Token(token_type, text, start_line, start_col)

    def _read_integer(self) -> Token:
        """Read an integer literal."""
        start_line, start_col = self.line, self.col
        result = []
        while True:
            ch = self._current()
            if ch is not None and ch.isdigit():
                result.append(self._advance())
            else:
                break
        return Token(TokenType.INTEGER, "".join(result), start_line, start_col)

    def _read_string(self) -> Token:
        """Read a double-quoted string literal with escape sequences."""
        start_line, start_col = self.line, self.col
        self._advance()  # consume opening quote
        result = []
        while True:
            ch = self._current()
            if ch is None:
                raise LexerError("Unterminated string literal", start_line,
                ↪  start_col)
            if ch == "\\":
                self._advance()  # consume backslash
                escape = self._current()
                if escape is None:
                    raise LexerError("Unterminated escape sequence", self.line,
                    ↪  self.col)
                self._advance()
                escape_map = {"n": "\n", "t": "\t", "\\": "\\", "\"": "\""}
                result.append(escape_map.get(escape, escape))
            elif ch == '"':
                self._advance()  # consume closing quote
                return Token(TokenType.STRING, "".join(result), start_line,
                ↪  start_col)
            else:
                result.append(self._advance())

```

```
    def _read_operator(self, start_ch: str, start_line: int, start_col: int) ->
    ↪ Token:
        """Read a single or multi-character operator."""
        # Check for two-character operators
        if start_ch == "=" and self._peek() == "=":
            self._advance()
            return Token(TokenType.EQEQ, "==", start_line, start_col)
        if start_ch == "!" and self._peek() == "=":
            self._advance()
            return Token(TokenType.NEQ, "!=", start_line, start_col)
        if start_ch == "<" and self._peek() == "=":
            self._advance()
            return Token(TokenType.LTE, "<=", start_line, start_col)
        if start_ch == ">" and self._peek() == "=":
            self._advance()
            return Token(TokenType.GTE, ">=", start_line, start_col)

        # Single-character operators
        op_map = {
            "+": TokenType.PLUS,
            "-": TokenType.MINUS,
            "*": TokenType.STAR,
            "/": TokenType.SLASH,
            "(": TokenType.LPAREN,
            ")": TokenType.RPAREN,
            "{": TokenType.LBRACE,
            "}": TokenType.RBRACE,
            ";": TokenType.SEMICOLON,
            ",": TokenType.COMMA,
            "=": TokenType.EQUALS,
            "<": TokenType.LT,
            ">": TokenType.GT,
        }
        token_type = op_map.get(start_ch)
        if token_type is None:
            raise LexerError(f"Unexpected character: {start_ch!r}", start_line,
            ↪  start_col)
        return Token(token_type, start_ch, start_line, start_col)

    def tokenize(self) -> List[Token]:
        """Tokenize the entire source and return a list of tokens."""
        tokens = []
        while True:
            self._skip_whitespace_and_comments()
            ch = self._current()
            if ch is None:
                tokens.append(Token(TokenType.EOF, "", self.line, self.col))
                break

            start_line, start_col = self.line, self.col
```

```
216
217             if ch.isalpha() or ch == "_":
218                 tokens.append(self._read_identifier())
219             elif ch.isdigit():
220                 tokens.append(self._read_integer())
221             elif ch == '"':
222                 tokens.append(self._read_string())
223             else:
224                 self._advance()
225                 tokens.append(self._read_operator(ch, start_line, start_col))
226
227         return tokens
228
229
230 def main():
231     """Demonstrate the lexer on sample code."""
232     source = '''
233     // Calculate factorial
234     if x > 0 {
235         return x * fact(x - 1);
236     } else {
237         return 1;
238     }
239     '''
240     lexer = Lexer(source)
241     tokens = lexer.tokenize()
242     for token in tokens:
243         print(token)
244
245
246 if __name__ == "__main__":
247     main()
```

Line-by-line explanation of key design decisions:

Lines 4-53: The TokenType enum defines every atomic unit the language recognizes. Each token type is explicit; there is no "OTHER" or "UNKNOWN" category. This precision matters because parsers depend on knowing exactly what tokens can appear in each position.

Lines 56-62: Every Token records its source location (line and column). This is not optional for production analyzers. When you report a finding to a developer, you must tell them exactly where it occurred. Source locations also enable IDE integration with clickable navigation to the problematic code.

Lines 78-91: The Lexer maintains position state (pos, line, col) and provides helper methods for reading ahead without consuming characters. This peek-and-consume pattern is fundamental to lexical analysis.

Lines 93-105: Whitespace and comment skipping happens between tokens, not within them. This keeps the tokenization logic clean. The implementation handles single-line comments (// style); a production lexer would also handle block comments (/* */), here-docs, and language-specific comment styles.

Lines 107-118: Identifier reading checks for alphanumeric characters and underscores, then looks up the result in a keyword table. This two-step approach is standard: first recognize the shape of an identifier, then classify it as a keyword or user-defined name based on its value.

Lines 135-154: String literal handling is more complex than it appears. The lexer must track escape sequences, detect unterminated strings, and report errors with accurate positions (the opening quote, not the end of file where the error was discovered).

Lines 156-180: Multi-character operator recognition uses lookahead to distinguish between = and ==, < and <=, etc. This pattern generalizes to any language with multi-character tokens that share prefixes: C++'s >> vs > >, Ruby's === vs ==, or Python's := vs :.

The tradeoff in this implementation is simplicity versus performance. A production lexer for a large language might use a precompiled finite automaton generated by tools like Flex or RE2, which can be significantly faster than character-by-character scanning. However, the hand-written approach gives you complete control over error messages and edge cases, which matters more for an analyzer than raw speed.

Parsing and Grammar Design

Once tokens are produced, parsing assembles them into a structured representation of the program: an abstract syntax tree (AST). The parser's job is to verify that the token sequence conforms to the language grammar and to build a tree that captures the hierarchical structure of the code.

Grammars define the rules for valid programs. A context-free grammar specifies how non-terminal symbols (like "expression" or "statement") can be composed from terminals (tokens) and other non-terminals. Here is a simple grammar for our expression language in Extended Backus-Naur Form (EBNF):

```
program       ::= statement*
statement     ::= if_statement
                | while_statement
                | return_statement
                | block
                | expression ";"
if_statement  ::= "if" "(" expression ")" statement ["else" statement]
while_statement ::= "while" "(" expression ")" statement
return_statement ::= "return" expression ";"
block         ::= "{" statement* "}"
expression    ::= comparison
comparison    ::= addition (("==" | "!=" | "<" | ">" | "<=" | ">=") addition)*
addition      ::= multiplication (("+" | "-") multiplication)*
multiplication ::= primary (("*" | "/") primary)*
primary       ::= NUMBER
                | STRING
                | IDENTIFIER
                | "(" expression ")"
                | IDENTIFIER "(" arguments? ")"
arguments     ::= expression ("," expression)*
```

This grammar encodes operator precedence: multiplication binds tighter than addition, which binds tighter than comparison. It also handles the structure of control flow statements and function calls. A parser built from this grammar will reject invalid programs like `if x { }` (missing parentheses around the condition) or `x = + 5` (incomplete left-hand side).

There are several approaches to implementing a parser:

Recursive descent parsing is hand-written code where each grammar rule becomes a function. It is intuitive, easy to debug, and gives you full control over error recovery. Most production parsers for specific languages use this approach or a variant.

Parser generators (YACC, Bison, ANTLR) take a grammar specification and generate parser code automatically. They handle complex parsing algorithms like LALR(1) or LL(*) that would be tedious to implement by hand. The tradeoff is that error messages can be cryptic, and debugging generated parsers is difficult.

Pratt parsing (top-down operator precedence parsing) is specialized for expression grammars. It handles operator precedence and associativity elegantly without requiring the grammar to enumerate all combinations explicitly.

Here is a recursive descent parser for our language:

```
from typing import List, Optional, Any, Dict


# AST Node Types
@dataclass
class Program:
    statements: List[Any]

@dataclass
class IfStatement:
    condition: Any
    then_branch: Any
    else_branch: Optional[Any] = None
    line: int = 0

@dataclass
class WhileStatement:
    condition: Any
    body: Any
    line: int = 0

@dataclass
class ReturnStatement:
    value: Any
    line: int = 0

@dataclass
class Block:
    statements: List[Any]
    line: int = 0

@dataclass
class ExpressionStatement:
    expression: Any
    line: int = 0

@dataclass
class BinaryOp:
    left: Any
    op: str
    right: Any
    line: int = 0

@dataclass
class NumberLiteral:
    value: int
    line: int = 0

@dataclass
class StringLiteral:
```

```
    value: str
    line: int = 0

@dataclass
class Identifier:
    name: str
    line: int = 0

@dataclass
class CallExpression:
    function: Any
    arguments: List[Any]
    line: int = 0


class ParseError(Exception):
    """Raised when the parser encounters invalid syntax."""
    def __init__(self, message: str, token: Token):
        super().__init__(f"Line {token.line}, col {token.col}: {message}")
        self.token = token


class Parser:
    """
    Recursive descent parser for the expression language.

    Each method corresponds to a grammar rule. The parser consumes tokens
    sequentially and builds an AST. On error, it reports the position of
    the unexpected token with a descriptive message.
    """

    def __init__(self, tokens: List[Token]):
        self.tokens = tokens
        self.pos = 0

    def _current(self) -> Token:
        return self.tokens[self.pos]

    def _peek(self, offset: int = 1) -> Token:
        idx = self.pos + offset
        if idx < len(self.tokens):
            return self.tokens[idx]
        return self.tokens[-1]  # EOF

    def _advance(self) -> Token:
        token = self.tokens[self.pos]
        if self.pos < len(self.tokens) - 1:
            self.pos += 1
        return token

```

```
    def _expect(self, token_type: TokenType, message: Optional[str] = None) ->
    ↪ Token:
        """Consume a token of the expected type or raise ParseError."""
        token = self._current()
        if token.type != token_type:
           msg = message or f"Expected {token_type.name}, got {token.type.name}"
            raise ParseError(msg, token)
        return self._advance()

    def parse(self) -> Program:
        """Parse the full program."""
        statements = []
        while self._current().type != TokenType.EOF:
            statements.append(self._parse_statement())
        return Program(statements)

    def _parse_statement(self) -> Any:
        """Parse a single statement."""
        token = self._current()

        if token.type == TokenType.IF:
            return self._parse_if_statement()
        elif token.type == TokenType.WHILE:
            return self._parse_while_statement()
        elif token.type == TokenType.RETURN:
            return self._parse_return_statement()
        elif token.type == TokenType.LBRACE:
            return self._parse_block()
        else:
            # Expression statement
            expr = self._parse_expression()
            self._expect(TokenType.SEMICOLON, "Expected ';' after expression")
            return ExpressionStatement(expr, token.line)

    def _parse_if_statement(self) -> IfStatement:
        """Parse: if (expr) stmt [else stmt]"""
        line = self._current().line
        self._advance()  # consume 'if'
        self._expect(TokenType.LPAREN, "Expected '(' after 'if'")
        condition = self._parse_expression()
        self._expect(TokenType.RPAREN, "Expected ')' after condition")
        then_branch = self._parse_statement()
        else_branch = None
        if self._current().type == TokenType.ELSE:
            self._advance()  # consume 'else'
            else_branch = self._parse_statement()
        return IfStatement(condition, then_branch, else_branch, line)

    def _parse_while_statement(self) -> WhileStatement:
        """Parse: while (expr) stmt"""
```

```
        line = self._current().line
        self._advance()  # consume 'while'
        self._expect(TokenType.LPAREN, "Expected '(' after 'while'")
        condition = self._parse_expression()
        self._expect(TokenType.RPAREN, "Expected ')' after condition")
        body = self._parse_statement()
        return WhileStatement(condition, body, line)

    def _parse_return_statement(self) -> ReturnStatement:
        """Parse: return expr;"""
        line = self._current().line
        self._advance()  # consume 'return'
        value = self._parse_expression()
        self._expect(TokenType.SEMICOLON, "Expected ';' after return value")
        return ReturnStatement(value, line)

    def _parse_block(self) -> Block:
        """Parse: { stmt* }"""
        line = self._current().line
        self._advance()  # consume '{'
        statements = []
        while self._current().type not in (TokenType.RBRACE, TokenType.EOF):
            statements.append(self._parse_statement())
        self._expect(TokenType.RBRACE, "Expected '}' to end block")
        return Block(statements, line)

    def _parse_expression(self) -> Any:
        """Parse comparison expressions (lowest precedence)."""
        return self._parse_comparison()

    def _parse_comparison(self) -> Any:
        """Parse comparisons with ==, !=, <, >, <=, >=."""
        left = self._parse_addition()
        while self._current().type in (
            TokenType.EQEQ, TokenType.NEQ,
            TokenType.LT, TokenType.GT,
            TokenType.LTE, TokenType.GTE
        ):
            op_token = self._advance()
            right = self._parse_addition()
            left = BinaryOp(left, op_token.value, right, op_token.line)
        return left

    def _parse_addition(self) -> Any:
        """Parse addition and subtraction."""
        left = self._parse_multiplication()
        while self._current().type in (TokenType.PLUS, TokenType.MINUS):
            op_token = self._advance()
            right = self._parse_multiplication()
            left = BinaryOp(left, op_token.value, right, op_token.line)
```

```
        return left

    def _parse_multiplication(self) -> Any:
        """Parse multiplication and division."""
        left = self._parse_primary()
        while self._current().type in (TokenType.STAR, TokenType.SLASH):
            op_token = self._advance()
            right = self._parse_primary()
            left = BinaryOp(left, op_token.value, right, op_token.line)
        return left

    def _parse_primary(self) -> Any:
        """Parse primary expressions: literals, identifiers, calls, parens."""
        token = self._current()

        if token.type == TokenType.INTEGER:
            self._advance()
            return NumberLiteral(int(token.value), token.line)
        elif token.type == TokenType.STRING:
            self._advance()
            return StringLiteral(token.value, token.line)
        elif token.type == TokenType.IDENTIFIER:
            self._advance()
            # Check for function call
            if self._current().type == TokenType.LPAREN:
                return self._parse_call(token.value, token.line)
            return Identifier(token.value, token.line)
        elif token.type == TokenType.LPAREN:
            self._advance()  # consume '('
            expr = self._parse_expression()
            self._expect(TokenType.RPAREN, "Expected ')' after expression")
            return expr
        else:
            raise ParseError(f"Unexpected token in expression:
            ↪ {token.type.name}", token)

    def _parse_call(self, func_name: str, line: int) -> CallExpression:
        """Parse function call: name(args)."""
        self._advance()  # consume '('
        arguments = []
        if self._current().type != TokenType.RPAREN:
            arguments.append(self._parse_expression())
            while self._current().type == TokenType.COMMA:
                self._advance()  # consume ','
                arguments.append(self._parse_expression())
        self._expect(TokenType.RPAREN, "Expected ')' after arguments")
        return CallExpression(Identifier(func_name, line), arguments, line)


def main():
```

```
    """Demonstrate parsing."""
    source = '''
    if x > 0 {
        return x * fact(x - 1);
    } else {
        return 1;
    }
    '''
    lexer = Lexer(source)
    tokens = lexer.tokenize()
    parser = Parser(tokens)
    tree = parser.parse()
    print(ast.dump(tree, indent=2))


if __name__ == "__main__":
    main()
```

Key design decisions in this parser:

Lines 14-67: Each AST node type is a dataclass capturing the essential structure of that construct. An IfStatement holds its condition, then-branch, optional else-branch, and source line. This explicit structure makes it easy for analysis passes to traverse and inspect the tree.

Lines 82-95: The Parser maintains a position in the token stream and provides methods for looking ahead and consuming tokens. The _expect method is critical: it validates that the input matches the grammar at each step and produces clear error messages when it does not.

Lines 107-130: Statement parsing uses a simple dispatch on the current token type. If we see 'if', we parse an if-statement; if we see '{', we parse a block; otherwise, we assume an expression statement. This pattern generalizes to any language: look at the leading token to determine which production to apply.

Lines 159-186: Expression parsing respects operator precedence through a hierarchy of methods. _parse_comparison calls _parse_addition, which calls _parse_multiplication, which calls _parse_primary. Each level handles its operators in a left-associative loop, then delegates to the next tighter-binding level. This structure directly mirrors the grammar and ensures correct precedence without explicit rules.

The tradeoff here is between flexibility and complexity. A recursive descent parser is straightforward for most language constructs but becomes unwieldy for languages with complex expression syntax (C++, Rust, or JavaScript with

their dozens of operators and special forms). For those cases, a Pratt parser or a grammar-based generator like ANTLR is more practical.

Error Recovery in Real Compilers

The parser above fails fast: the first syntax error it encounters raises an exception and stops. This is fine for a standalone analyzer that processes one file at a time, but production compilers and IDE-integrated analyzers must do better. When you are analyzing a 50,000-line file, you want to report all the errors, not just the first one. And when you are providing real-time feedback in an editor, the user has typed something incomplete; stopping at the first error would make the tool useless.

Error recovery is the technique of continuing parsing after encountering an error, skipping enough tokens to resynchronize with a point where valid parsing can resume, and then continuing to find more errors. There are several strategies:

Panic mode recovery skips tokens until finding a "synchronizing token" that is likely to mark the start of a new statement or construct. Common synchronizing tokens include semicolons, closing braces, and keywords like `if`, `while`, and `return`. This is simple but can skip over multiple real errors if they cluster together.

Phrase-level recovery attempts to fix the error locally by inserting, deleting, or replacing a token. For example, if a semicolon is missing at the end of a statement, the parser might pretend it was there and continue. This produces more accurate error reporting but requires careful design to avoid cascading false errors.

Global correction searches for the minimal edit distance between the erroneous input and some valid program. This is theoretically ideal but computationally expensive, so it is rarely used in practice except in specialized tools like spell checkers.

Here is how panic mode recovery would be integrated into our parser:

```
    def _parse_with_recovery(self) -> Program:
        """Parse with error recovery: report all errors, not just the first."""
        statements = []
        errors = []

        while self._current().type != TokenType.EOF:
            try:
                stmt = self._parse_statement()
                statements.append(stmt)
            except ParseError as e:
                errors.append(e)
                self._recover_from_error()

        return Program(statements), errors

    def _recover_from_error(self):
        """Skip tokens until we find a synchronizing point."""
        sync_tokens = {
            TokenType.SEMICOLON,
            TokenType.RBRACE,
            TokenType.IF,
            TokenType.ELSE,
            TokenType.WHILE,
            TokenType.RETURN,
            TokenType.LBRACE,
        }
        while self._current().type not in sync_tokens and self._current().type
        ↪  != TokenType.EOF:
            self._advance()
```

The tradeoff is precision versus robustness. Panic mode recovery ensures the parser always completes, but the AST it produces for erroneous code may be structurally incorrect, which can confuse downstream analysis passes. A production analyzer must handle this gracefully, either by marking unreliable portions of the tree or by degrading to simpler analyses when the parse is suspect.

Why You Need This Background as an Analyst

You might be wondering whether you really need to understand lexing and parsing if you plan to use existing tools like ESLint, Pylint, or Semgrep. The answer depends on how deep you want to go.

If you only need to write rules for existing analyzers, you can work entirely at the AST level and never touch a lexer or parser. But if you want to build

custom analyzers, understand why certain analyses are expensive, debug false positives, or analyze languages that lack mature tooling, this background is essential.

Specifically, understanding the compiler pipeline helps you answer these practical questions:

What information is available at each stage? Lexical analysis gives you tokens with positions. Parsing gives you structure (AST). Semantic analysis gives you types and symbols. You cannot perform type-based analysis before types are known; you cannot reason about control flow before the AST exists.

Why do some analyses scale poorly? Interprocedural analyses require building call graphs, which requires resolving all function references, which requires parsing all files and constructing symbol tables. Each step adds complexity and memory overhead. Understanding the pipeline helps you estimate costs and design incremental strategies.

Where do false positives come from? Many false positives arise from incomplete semantic information: the analyzer cannot resolve a reference because it lacks context, so it assumes the worst case. Knowing which phase provides which information helps you understand the root cause of imprecision.

How do I analyze a new language? If your team uses a language without good static analysis tooling, understanding lexing and parsing tells you what infrastructure to build first: a lexer, then a parser, then semantic analysis, then the specific checks you need.

The next chapter builds directly on this foundation by exploring abstract syntax trees in depth: how they are structured, how to traverse them efficiently, and how to use them as the basis for static analysis.

Chapter 3: Abstract Syntax Trees and Code Representation

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

What an AST Is (and Is Not)

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

Building an AST from Tokens

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

The Visitor Pattern for Tree Traversal

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

Implementing a Concrete AST Visitor

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

Tree Transformations and Rewrites

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

Comparing ASTs: Diffing Generated Code

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

ASTs as the Foundation for Analysis

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

Chapter 4: Control Flow Graphs and Program Structure

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

From AST to Control Flow Graph

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

Basic Blocks and Dominators

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

Reachability Analysis

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

Detecting Dead Code in AI Output

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

Loop Analysis and Termination Questions

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

Exception Flow and Non-Local Control

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

Why CFGs Matter for AI Code Analysis

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

Chapter 5: Data Flow Analysis and Symbol Resolution

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

The Data Flow Problem

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

Forward vs Backward Analysis

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

Reaching Definitions

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

Live Variable Analysis

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

Symbol Tables and Scope Resolution

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

Call Graph Construction

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

Why Data Flow Matters for AI Code Analysis

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

Chapter 6: Type Systems and Type-Based Analysis

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

What Types Are For

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

Static vs Dynamic Typing for Analysis

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

Type Inference Basics

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

Detecting Type Confusion in Generated Code

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

Null Safety and Option Types

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

Generics, Polymorphism, and AI Pitfalls

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

Why Type-Based Analysis Matters for AI Code

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

Chapter 7: Intermediate Representations and Cross-Language Analysis

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

Why We Need Intermediate Representations

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

LLVM IR as a Case Study

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

Three-Address Code and SSA Form

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

Tree-Sitter: Parsing Anything

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

Building a Language-Agnostic Analyzer

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

Cross-Language Call Graphs

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

Why IRs Matter for AI Code Analysis

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

Chapter 8: Taint Analysis and Security Vulnerabilities

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

The Taint Model

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

Sources, Sinks, and Sanitizers

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

Implementing a Taint Tracker

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

Prompt-Induced Vulnerabilities

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

Injection Attacks in Generated Code

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

Insecure Default Patterns from AI

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

Why Taint Analysis Matters for AI Code

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

Chapter 9: Symbolic Execution and Path-Sensitive Analysis

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

Concrete vs Symbolic Execution

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

Path Conditions and SMT Solvers

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

Building a Basic Symbolic Executor

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

Path Explosion and Mitigation

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

When Symbolic Execution Is Worth It

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

Concolic Testing for AI Code

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

Why Symbolic Execution Matters for AI Code

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

Chapter 10: Abstract Interpretation and Sound Analysis

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

The Abstract Interpretation Framework

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

Lattices, Join, and Meet

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

Widening and Narrowing

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

Interval Analysis

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

Pointer Analysis

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

Tradeoffs: Precision vs Performance

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

Why Abstract Interpretation Matters for AI Code

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

Chapter 11: Rule Engines and Custom Linting

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

The Anatomy of a Static Analysis Rule

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

Pattern Matching on ASTs

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

Building a Rule Engine from Scratch

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

Semgrep-Style Structural Search

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

CodeQL Query Language Design

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

Managing Thousands of Rules

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

Why Rule Engines Matter for AI Code

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

Chapter 12: AI-Specific Code Smells and Detection Strategies

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

Hallucinated APIs and Nonexistent Functions

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

Inconsistent Coding Patterns and Style Drift

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

Dead Code and Redundant Logic

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

Hidden Dependencies and Missing Imports

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

Context Window Artifacts

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

Incomplete Implementations and TODO Sprawl

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

Duplicated Logic Across Files

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

Over-Engineering and Unnecessary Complexity

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

Why Detecting AI Code Smells Matters

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

Chapter 13: Building a Production Static Analyzer

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

Architectural Decisions for Scale

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

Incremental Analysis and Caching

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

Parallelization Strategies

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

False Positive Management

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

SARIF and Standardized Output

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

Integration with CI/CD Pipelines

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

Why Production Architecture Matters

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

Chapter 14: The Tooling Ecosystem

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

Language-Specific Analyzers

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

Roslyn (C# and Visual Basic)

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

Clang Static Analyzer (C, C++, Objective-C)

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

ESLint (JavaScript and TypeScript)

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

Pylint (Python)

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

Rust Clippy

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

Query-Based Tools (CodeQL, Semgrep)

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

CodeQL

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

Semgrep

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

IDE Integration Strategies

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

Analyzing Large AI-Assisted Codebases

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

Choosing Your Stack

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

Building on Existing Infrastructure

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

Chapter 15: Defense in Depth — Integrating Static Analysis into AI Workflows

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

Pre-Generation Guardrails

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

Post-Generation Validation Pipelines

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

Developer Feedback Loops

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

Measuring AI Code Quality

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

The Future of Static Analysis and AI

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

A Practical Checklist

Conclusion

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

References

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