Overview

1. Language Design

Hanami provides a one-to-one mapping between its "garden" vocabulary and canonical C++ constructs—e.g. `garden` → `namespace`, `species` → `class`, `bloom`/`water` → stream operators. Because all native C++ types, operators, and literals remain valid, users can freely intermix Hanami keywords with regular C++ expressions without extra lowering logic. The shared `TokenType` enumeration ensures every compiler phase agrees on the same keyword set, preventing drift between stages.

2. Compiler Pipeline

Stage	**Implementation Highlights – What the Stage Does**	**Output**
Lexer – Lexical Analysis	• Reads the character stream once and groups into tokens (names, numbers, strings, operators, Hanami keywords). • Ignores single-line and multi-line comments; records exact line/column positions for later error reporting. • Detects and reports early errors like invalid characters, unclosed strings, etc.	Token list with positions
Parser – Recursive-Descent Syntax Analysis	• Receives tokens and builds an Abstract Syntax Tree (AST) representing program structure (gardens, species, functions, statements). • Uses manual recursive functions, supports panic-mode recovery: on syntax error, skips to synchronization point to continue catching more errors instead of stopping.	Serializable AST (JSON)

Semantic Analysis + IR	• Traverses AST to build symbol tables, apply scoping and visibility rules (open/hidden/guarded), perform static type checks (e.g. disallow `blossom` in `void` functions). • Labels each AST node with type and ID, emits language-neutral Intermediate Representation (JSON).	Rich-context IR (JSON)
Code Generation / Transpiler	• Consumes the unified IR and, via pluggable visitors, generates source code in C++, Java, Python, and JavaScript. • Ensures independent target generation: failure in one target does not affect others. • Retains original line numbers so runtime errors in generated code can be traced back to Hanami source.	Files: `.cpp`, `.java`, `.py`, `.js`