# Hanami: A Nature-Inspired, Multi-Target Transpilation Programming Language

Ngo Thanh Trung
Troy University
Hanoi, Vietnam
tngo220196@troy.edu

Pham Tien Dat
Troy University
Hanoi, Vietnam
dpham220298@troy.edu

Pham Thai Duong
Troy University
Hanoi, Vietnam
dpham220299@troy.edu

Bui Dang Quoc An
Troy University
Hanoi, Vietnam
abui220008@troy.edu

***Abstract*—This paper introduces Hanami, a transpiled programming language developed as an educational project in CS3370 - Nature of Programming Languages. Designed to explore programming language fundamentals, Hanami features a low coupling models with input/output system, effective error reporting mechanism, and intuitive syntax inspired by natural elements. Rather than targeting machine code directly, our transpiler generates equivalent programs in C++, Java, JavaScript, and Python, providing cross-platform compatibility while simplifying implementation. The paper details our design philosophy, compiler pipeline implementation, and the development of a web-based interactive playground that makes the language immediately accessible. Through this project, we demonstrate how theoretical language concepts can be translated into a practical educational tool, offering insights into lexical analysis, parsing, semantic processing, and code generation. Hanami exemplifies how programming language education can bridge theory and practice through hands-on implementation experience.**

***Index Terms*—programming language design, compiler construction, transpiler, multi-target transpilation, type systems, error handling systems, modular architecture, code generation, recursive descent parsing, web-based development tools**

CONTENTS

## I. Introduction

Programming languages play a crucial role in software development, each designed with specific goals and trade-offs. As students in CS3370 - Nature of Programming Languages, we've been exploring these concepts through the development of Hanami, a transpiled programming language created as an educational project.

Modern programming involves working with various languages, each with distinct characteristics: Java offers portability, C++ prioritizes performance, Rust focuses on memory safety, and Python emphasizes readability. Through our coursework, we recognized how these trade-offs impact development choices and wondered if a student project could explore alternative approaches.

### A. Motivation and Educational Objectives

Our motivation for creating Hanami stems from three key observations in our programming language studies:

- Type systems typically follow either static or dynamic approaches, with few options in between
- Programming languages often make fundamental design choices that limit their flexibility across different use cases
- Learning compiler/transpiler design principles requires hands-on experience with language implementation

Hanami addresses these educational objectives through a unique approach - instead of compiling to machine code, it transpiles to established languages including C++, Java, JavaScript, and Python. This design decision allows us to focus on language design principles while leveraging existing compilation infrastructure.

The name "Hanami", inspired by the Japanese tradition of flower viewing, reflects our design philosophy. Just as hanami celebrates the beauty of cherry blossoms, our language aims to showcase the elegant aspects of programming language design we've studied in this course.

### B. Educational Impact and Academic Significance

The development of Hanami provides several significant educational benefits:

- **Practical Application of Theory**: Students gain first-hand experience implementing theoretical concepts from programming language design.
- **Cross-language Understanding**: The transpilation approach deepens understanding of how different languages implement similar concepts.
- **Compiler Construction Knowledge**: Each phase of development reinforces compiler construction principles in a concrete, accessible way.
- **Design Trade-off Analysis**: Students make deliberate design choices, weighing the consequences of each decision.

This project bridges the gap between theoretical knowledge and practical implementation, allowing students to experience the challenges and complexities of language design that are difficult to appreciate through traditional coursework alone.

TABLE I: Hanami Keyword Examples (Intro)

| **Hanami** | **C++ Equivalent** | **Meaning** |
|---|---|---|
| garden <Name> | namespace <Name> | Declares a namespace |
| species <Name> | class <Name> | Declares a class |
| grow <Name>() -> <type> | <type> <Name>() | Declares a function |
| bloom << x; | std::cout << x; | Prints to console |

### C. Technical Overview and Implementation Approach

Key features of our implementation include:

- A lexical analyzer (lexer) that converts source code into tokens
- A parser that builds an abstract syntax tree (AST) from these tokens
- A semantic analyzer that verifies program correctness
- A transpiler that converts Hanami code to target languages

Hanami's syntax draws inspiration from multiple languages while introducing nature-themed keywords aligned with modern programming language design principles [1]. For instance, "garden" replaces "namespace," "species" defines classes, and functions are declared using "grow." This design choice makes the language both distinctive and memorable while serving as a practical demonstration of syntax design principles.

### D. Innovative Features and Distinguishing Characteristics

Hanami incorporates several innovative features that distinguish it from typical student projects:

- **Multi-target Transpilation**: Unlike most educational language projects that target a single platform, Hanami transpiles to multiple languages, enabling cross-platform compatibility.
- **Nature-Inspired Lexical Design**: The consistent nature theme provides a cohesive and memorable programming experience while demonstrating naming convention principles.
- **Graduated Type System**: Rather than adopting a purely static or dynamic approach, Hanami implements a type system with configurable strictness levels, allowing students to explore the spectrum between these paradigms.
- **Effect Handling**: Inspired by modern research in programming languages, Hanami includes basic effect handling mechanisms that allow for more controlled side effects.

These features were carefully selected to balance educational value with implementation feasibility while showcasing important concepts in modern programming language design.

### E. Paper Organization

Through this project, we've gained practical insights into compiler construction stages including lexical analysis, parsing, semantic analysis, and code generation. Our implementation demonstrates these concepts while creating a functional transpiler that converts Hanami code to multiple target languages.

The remainder of this paper is organized as follows: Section II details Hanami's language design journey and features. Section III describes our transpiler implementation, including

the architecture and pipeline stages. Section IV discusses the evaluation process, limitations, and potential future work. Section VI summarizes our findings.

This project represents our practical exploration of programming language concepts and compiler design principles learned throughout the CS3370 course.

## II. Language Design Journey and Core Features

### A. Initial Vision: Toward a Revolutionary Language

Our journey began with an ambitious vision: to create a programming language that would revolutionize software development similar to how Python transformed the industry decades ago. Python succeeded through prioritizing extremely high-level design, readable and simple syntax, consistent design principles, and a "developer-first" philosophy that made programming more accessible and productive.

"What if we could create a language that addresses modern computing challenges while maintaining that same human-centric approach?" we asked ourselves. The team envisioned a language that would combine the readability of Python, the performance of Rust, the type safety of TypeScript, and the versatility of C++ - truly a language for the modern era.

We initially explored several revolutionary concepts in language design, following methodologies similar to those outlined in transpiler research [2]–[4].

Listing 1: Early concept sketch for declarative parallel processing

```
# Early concept sketch for declarative
parallel processing
parallel compute(data) {
    partition(data, optimal)
    map(transform_function)
    reduce(combine_function)
}
```

This early syntax draft aimed to simplify parallel computing by abstracting away thread management while maintaining readable code. Similarly, we explored novel approaches to memory management, context-aware computation, and intuitive concurrency primitives.

### B. Practical Constraints and Redirected Ambitions

Reality soon set in. Creating a fully-featured language with compiler infrastructure would require resources far beyond our scope as a student team in CS3370. As one team member memorably put it: "We wanted to build a rocket ship, but we barely have time to build a bicycle."

Our next approach was to combine existing languages. Perhaps we could merge C++'s performance with Java's portability? Or Python's readability with JavaScript's ubiquity? We quickly encountered fundamental incompatibilities in these approaches:

- **Memory Management Conflicts:** Reconciling garbage collection with manual memory management proved conceptually challenging.
- **Type System Contradictions:** Marrying static typing with dynamic typing created more complications than solutions.
- **Paradigm Dissonance:** Functional, object-oriented, and procedural paradigms fought for dominance in our hybrid designs.

These technical challenges were compounded by a growing realization: our hybrid language risked becoming a confusing "language soup" rather than a cohesive system. As Prof. Han had warned in our course readings, "Simplicity of design is paramount."

### C. Conceptual Exploration and Identity Formation

Refocusing our efforts, we began a series of creative brainstorming sessions to find our language's identity. We explored various themes that might inform our design:

- A language with animal-themed keywords (cat&dogs, where return → fetch, function → throw).
- A Vietnamese programming language where we would translate every single syntax of a high-level language to our native language.
- A music-themed language where programs would read like musical notation.
- A project explicitly designed for educational purposes, by just re-designing C++ from scratch.

We decided to let fate decide which is the better idea, using the popular duck race game, we have chosen the winner. The adoration and enthusiasm about Japanese aesthetics of our team leader - Trung has been answered, and we coined upon the concept of "hanami" - the tradition of flower viewing.

What if our language embodied the principles of hanami - appreciating beauty, embracing simplicity, and celebrating transience? This conceptual frame resonated with our team and provided the metaphorical foundation for our design choices.

Listing 2: Example Hanami code showing nature-inspired syntax

```
// Example Hanami code showing
nature-inspired syntax
garden NatureDemo {
    species Tree {
        seed height: float;
        seed age: int;

        grow calculateGrowth() -> float {
            return this.height * 0.1 *
            this.age;
        }
    }

    grow main() -> void {
        Tree oak;
        oak.height = 10.5;
        oak.age = 20;
        bloom << "Growth rate: " <<
        oak.calculateGrowth();
    }
}
```

Just as hanami celebrates the temporary beauty of cherry blossoms, our language would acknowledge its role as an educational vessel - a means to understand language design rather than a commercial product.

### D. The Hanami Philosophy: Design Principles

From our conceptual explorations emerged core design principles that would guide our implementation:

- **Aesthetic Clarity:** Code should be visually appealing and immediately readable.
- **Conceptual Harmony:** Language features should work together coherently.
- **Educational Transparency:** The language should reveal rather than hide its mechanisms.
- **Practical Efficiency:** Development should focus on achievable goals with maximum learning value.

We faced a critical implementation decision: should we build a compiler generating machine code, or take another approach? Traditional compiler development would require creating a complete toolchain with lexer, parser, optimizer, and code generator for a specific machine architecture.

After weighing our options against our educational goals, we made a pivotal choice: Hanami would be a *transpiled* language. Rather than generating machine code, it would transform into established languages like C++, Java, JavaScript, and Python. This approach offered several advantages:

- We could focus on language design without getting lost in low-level implementation details.
- Students could observe how language features map to different target languages.
- The output would be immediately usable on any platform supporting the target languages.
- We could concentrate our learning on the front-end aspects of language implementation.

This decision shaped our entire development approach, aligning with source-to-source compilation strategies [5]–[7].

### E. Syntax and Semantics: The Aesthetic of Code

Hanami's syntax embraces its nature-inspired identity while maintaining familiarity for students of mainstream languages. We deliberately created a one-to-one mapping between Hanami constructs and conventional programming constructs to facilitate learning.

The language employs a consistent "garden" metaphor throughout:

TABLE II: Hanami Syntaxes Ideas

| Hanami Construct | Metaphorical Meaning |
|---|---|
| garden | A collection of related elements |
| species | A template for creating objects |
| seed | The internal data of an object |
| grow | Operations that develop over time |
| bloom | Presenting results to the world |
| water | Nourishing the program with data |
| root | Drawing from established foundations |

We maintained C++ syntax for expressions, control flow, and literals to ensure familiarity while introducing our metaphor-driven keywords. This created a language that feels both novel and approachable:

Listing 3: Hanami example with Complex numbers

```
garden Mathematics {
    species Complex {
        seed real: float;
        seed imaginary: float;

        grow add(other: Complex) -> Complex {
            Complex result;
            result.real = this.real +
            other.real;
            result.imaginary = this.imaginary
            + other.imaginary;
            return result;
        }

        grow toString() -> string {
            // Note: String concatenation
            might need a helper in actual
            implementation
            // This is illustrative
            return std::to_string(this.real)
            + " + " +
            std::to_string(this.imaginary) +
            "i";
        }
    }
}
```

### F. Core Language Features

*1) Growing in many soils:* Central to Hanami's design is its multi-target transpilation capability. Unlike traditional compilers that generate machine code, our transpiler transforms Hanami source code into equivalent programs in C++, Java, JavaScript, and Python.

We approached transpilation through an intermediate representation (IR) that captures the essential semantics of a Hanami program independent of any target language. This design allows each language backend to be developed independently:

$$\text{Hanami Source} \rightarrow \text{Lexer} \rightarrow \text{Parser} \rightarrow \text{AST} \rightarrow$$

$$\text{Semantic Analysis} \rightarrow \text{IR} \rightarrow \text{Target Code Generators}$$

The challenging aspects of transpilation include:

- Mapping Hanami's static type system to both static and dynamic target languages.
- Preserving object-oriented semantics across languages with different object models.
- Handling language-specific features like memory management and exceptions.

For example, when transpiling Hanami's object system to JavaScript (which uses prototypal inheritance), we implemented a class emulation layer that preserves the semantics of Hanami's class-based system.

*2) Type System:* Hanami implements a gradual type system inspired by modern typing paradigms [8]–[10]. The type system includes:

- Primitive types: `int`, `float`, `double`, `char`, `boolean`, `string`.
- Composite types: arrays, `species` (classes), enums.

- Type safety with compile-time checking.

Our type system design faced an interesting challenge: how to maintain static typing when transpiling to dynamically-typed languages like JavaScript and Python. We addressed this through runtime type checking in the generated code:

Listing 4: Python transpilation output with runtime type checking

```
# Python transpilation output with runtime
type checking
def add(x, y):
    # Type checking resembling Hanami's
    static checks
    if not isinstance(x, int) or not
    isinstance(y, int):
        raise TypeError("Arguments must be
        integers")
    return x + y
```

This approach preserves the semantics of Hanami's type system across all target languages while providing students insight into the differences between static and dynamic typing.

*3) Function System:* Hanami's function system combines familiar elements with our nature-inspired syntax. Functions are declared using the `grow` keyword, with a return type specified after an arrow:

Listing 5: Hanami Fibonacci function example

```
grow fibonacci(n: int) -> int {
    if (n <= 1) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

The function system supports:

- Named parameters with explicit types.
- Return type declarations.
- Function overloading based on parameter types.
- Member functions within `species` (classes).
- Anonymous functions (lambdas).

Our design process for the function system involved balancing familiarity (using established parameter and return type syntax) with our metaphorical theme (using "grow" to represent operations that develop and produce results).

*4) Object-Oriented Features:* Hanami's object-oriented programming model centers around the "species" concept (equivalent to classes). This metaphor extends naturally to inheritance ("root"), instantiation ("planting"), and method definition ("growing").

Listing 6: Hanami OOP example with inheritance

```
species Animal {
    seed name: string;
    seed age: int;

    grow speak() -> void {
        bloom << "Generic animal sound";
    }
}; // Added semicolon for C++ style
consistency
```

```
species Dog root Animal {
    grow speak() -> void {
        bloom << "Woof! My name is " <<
        this.name;
    }
}; // Added semicolon
```

The OOP system includes:

- Encapsulation through access modifiers (public, private, protected).
- Inheritance with the "root" keyword.
- Polymorphism through method overriding.
- Constructors and destructors.

Our implementation prioritizes clean, understandable OOP semantics that translate well to our target languages while maintaining the nature metaphor that defines Hanami's identity.

### G. Reflections on Language Design

The development of Hanami offered valuable insights into programming language design principles. We discovered that even seemingly simple design choices often involve complex trade-offs between:

- Expressiveness versus simplicity.
- Familiarity versus innovation.
- Metaphorical consistency versus practical utility.
- Educational value versus implementation complexity.

By focusing on transpilation rather than full compilation, we were able to explore language design concepts more fully while still producing a functional system. Our nature-inspired syntax created a memorable, cohesive language identity while maintaining the functionality expected in a modern programming language.

## III. Implementation: System Architecture and Pipeline

Developing Hanami has helped us understand the stages involved in designing and building a programming language. Our transpiler includes the following components: Lexer, Parser, Semantic Analyzer, and Code Generator. Each component was a valuable learning opportunity about compiler design and programming language principles.

### A. High-Level Architecture Overview

The Hanami compiler transforms Hanami source code (.hanami) into other languages like C++, Java, Python, or JavaScript. The compilation process involves distinct phases, each performing specific transformations.

*1) Architectural Model:* The Hanami compiler employs a modular pipeline architecture where each phase performs a specific transformation on the program representation, passing the result to the next phase. This design enables:

- **Separation of concerns**: Each component focuses on a well-defined task.
- **Extensibility**: New language features or target platforms can be added with minimal changes to other components.
- **Maintainability**: Components can be developed, tested, and modified independently.

- **Parallelization**: Some phases could potentially be executed concurrently.

*2) Data Flow Model:* The progression of program representation throughout the compilation process follows a series of well-defined transformations:

- Hanami Source Code (.hanami)
- Lexer → Token Stream
- Parser → Abstract Syntax Tree (AST)
- Semantic Analyzer → Annotated AST
- Code Generator → C++, Java, Python, or JavaScript Code

Each transformation preserves the semantic meaning of the program while changing its representation to facilitate subsequent processing steps.

### B. Compiler Pipeline Stages

The core pipeline consists of Lexical Analysis, Parsing, Semantic Analysis, and Code Generation. Each stage performs a specific function in transforming Hanami source code into target languages.

TABLE III: Compiler Pipeline Stages

| Stage | Implementation Highlights | Output |
|---|---|---|
| Lexer – Lexical Analysis | • Reads character stream once, groups into tokens<br>• Ignores comments; records line/column positions<br>• Detects early errors (invalid chars, unclosed strings) | Token list w/ positions |
| Parser – Recursive-Descent Syntax Analysis | • Receives tokens, builds Abstract Syntax Tree (AST)<br>• Uses manual recursive functions, supports recovery | Serializable AST (JSON) |
| Semantic Analysis + IR | • Traverses AST, builds symbol tables, type checks<br>• Labels AST nodes with type/ID, emits IR (JSON) | Rich-context IR (JSON) |
| Code Gen / Transpiler | • Generates C++, Java, Python, JavaScript<br>• Ensures independent target generation<br>• Retains line numbers for error tracing | Files: `.cpp`, `.java`, `.py`, `.js` |

### C. Lexer

The lexer converts the raw source code into a stream of tokens. This component is implemented in the `lexer/` directory.

*1) Lexer Components:*

- **TokenDefinition**: Defines the set of token types recognized by the Hanami language.
- **Tokenizer**: Implements the scanning logic to identify tokens in the source code.
- **ErrorReporter**: Reports lexical errors with precise source location information.

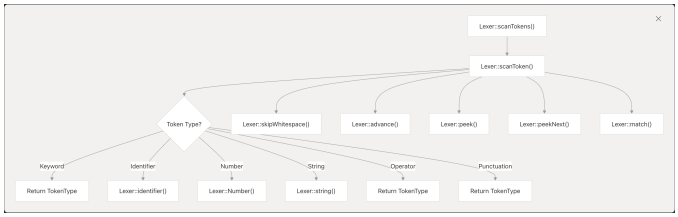



Fig. 1: Lexer Implementation

*2) Lexer Algorithm:* The lexer employs a deterministic finite automaton (DFA) approach following classical compiler design principles [1], [4].

1) Read input character by character.
2) Maintain current state based on previous characters.
3) Transition between states based on character class.
4) When an accepting state is reached, emit the corresponding token.
5) Handle error states with appropriate error messages.

### D. Parser

The parser constructs an Abstract Syntax Tree (AST) from the token stream, enforcing the syntactic rules of the Hanami language. This component is implemented in the `parser/` directory.

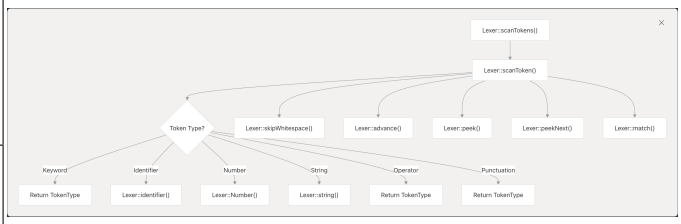



Fig. 2: Parser Implementation

*1) Parser Components:*

- **GrammarDefinition**: Formal definition of the Hanami language grammar.
- **Parser**: Implementation of parsing algorithms (recursive descent with predictive parsing).
- **ASTNodes**: Hierarchy of node types representing program constructs.
- **SyntaxErrorHandler**: Reports syntax errors with relevant context.

*2) Parsing Algorithm:* The parser uses a recursive descent approach with predictive parsing as demonstrated in compiler frameworks [11], [12].

1) Start with the top-level grammar rule.
2) For each non-terminal in the rule, recursively apply its production rules.
3) Match terminal symbols against the current token.
4) Construct AST nodes as grammar rules are successfully matched.
5) Handle synchronization points for error recovery.

*3) AST Structure:* The AST is structured as a hierarchical composition of nodes representing different program constructs, including expressions, statements, declarations, and type specifications.

### *E. Semantic Analyzer*

The semantic analyzer verifies the semantic correctness of the program and enriches the AST with type and scope information. This component is implemented in the `semantic_analyzer/` directory.

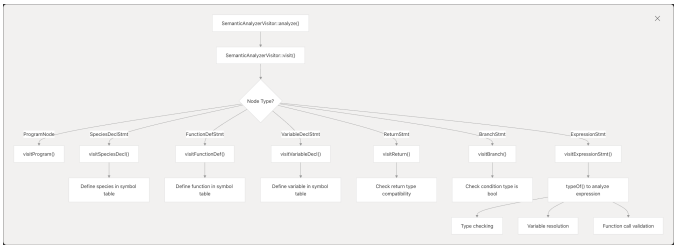



Fig. 3: Semantic Analyzer Implementation

*1) Semantic Analyzer Components:*

- **SymbolTable**: Manages symbols (variables, functions, types) and their attributes.
- **TypeSystem**: Defines and enforces the type rules of the Hanami language.
- **ScopeManager**: Tracks nested scopes and symbol visibility.
- **SemanticErrorReporter**: Reports semantic errors with contextual information.

*2) Symbol Table Structure:* The symbol table uses a hierarchical structure to represent nested scopes, allowing for efficient symbol lookup and scope management.

*3) Type System Details:* The type system defines primitive types and compound types using formal verification techniques [13]–[15].

- **Primitive Types**: int, double, float, bool, char, etc.
- **Compound Types**: arrays, structures, enums.
- **Type Relationships**: compatibility, conversion rules.
- **Type Checking**: assignment compatibility, operator operand verification.

*4) Semantic Analysis Process:*

1) Traverse the AST using the visitor pattern.
2) Build and populate the symbol table.
3) Perform type checking and inference.
4) Validate semantic constraints (e.g., no duplicate declarations).
5) Annotate the AST with semantic information.

### *F. Code Generator*

The code generator translates the semantically validated AST into target languages such as C++, Java, Python, or JavaScript. This component is implemented in the `codegen/` directory.

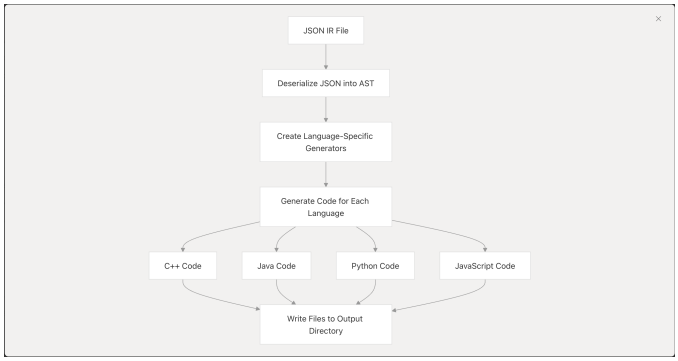



Fig. 4: Code Generation Workflow

*1) Code Generator Components:*

- **ASTVisitor**: Traverses the annotated AST.
- **CodeEmitter**: Generates code for the target language.
- **LanguageSpecificEmitter**: Handles specific syntax and semantics of the target language.

*2) Code Generation Process:* The code generation process involves:

- Traverse the AST.
- Generate code for each node, considering the target language.
- Handle language-specific features and mappings.
- Output the generated code.

*3) Common Patterns in Code Generators:* All language-specific generators share similar structure and patterns:

- Maintain a stringstream for accumulating generated code
- Track language-specific imports/includes
- Implement language-specific type mapping
- Implement language-specific operator mapping
- Handle language-specific syntax and semantics when visiting AST nodes

*4) Supported Target Languages:* The code generator currently supports:

- C++
- Java
- Python
- JavaScript

### *G. Support Systems*

*1) Build System:* The build system orchestrates the compilation process, managing dependencies and build configurations. It typically includes:

- **Makefile/Build Script**: Defines build targets and dependencies.
- **Build Configuration**: Manages compiler flags and options.
- **Dependency Tracking**: Determines which files need recompilation.

Common build targets might include ‘clean‘, ‘build‘, ‘test‘, ‘install‘, and ‘package‘. Build configurations often support Debug, Release, cross-compilation, and feature toggles.

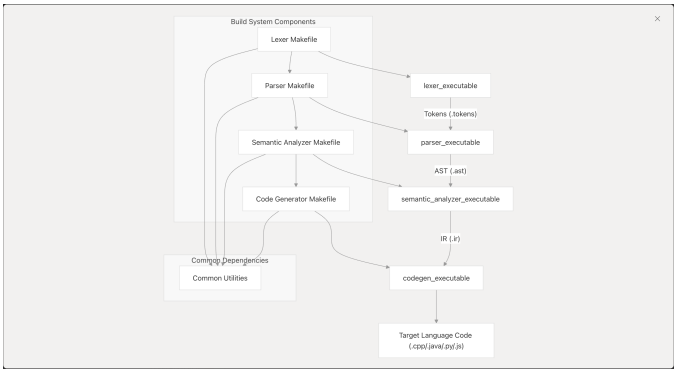



Fig. 5: Build System Overview

**Cross-Platform Support**: The build system provides cross-platform support for both Windows and Unix-like operating systems:

Uses $(OS) variable to detect Windows Handles path differences between platforms Adjusts deletion commands (del vs rm -f) Handles output path formatting differences

*2) Common Utilities:* The `common/` directory contains shared utilities used across compiler components.

- **Error Handling**: Consistent error reporting infrastructure.
- **Source Location Management**: Tracking file, line, and column information.
- **Memory Management**: Custom allocators or strategies.
- **Diagnostics**: Error and warning message formatting.
- **Data Structures**: Specialized structures like Symbol Tables, AST nodes, String Interning pools.
- **I/O Facilities**: For source reading, output generation, error streams.

### H. Inter-Component Communication

*1) Data Exchange Formats:* Components communicate through well-defined data structures:

- **Token Stream**: Between lexer and parser.
- **Abstract Syntax Tree (AST)**: Between parser and semantic analyzer.
- **Annotated AST**: Between semantic analyzer and code generator.
- **Target Language Code**: Output from code generator.

*2) Error Handling Strategy:* The compiler implements a unified error handling strategy:

- **Error Categories**: Lexical, syntactic, semantic, code generation.
- **Error Severity**: Fatal, error, warning, note.
- **Error Context**: Source location, relevant symbols, suggested fixes.
- **Error Recovery**: Mechanisms to continue compilation after errors where possible.

*3) Progress Tracking:* The compiler can track compilation progress for reporting:

- Phase completion indications.
- Statistics (time, memory usage).
- Progress indicators for long operations.

## IV. Evaluation

During Hanami's development, we evaluated the language's functionality and the transpiler's correctness through several simple program examples.

### A. Evaluation Approach

Our evaluation focused on:

- Basic algorithms (sorting, searching, string processing).
- Small programs demonstrating OOP (e.g., student management system, simple simulations).
- Basic file handling and I/O operations.

Testing involved writing Hanami code, transpiling it to each target language (C++, Java, Python, JavaScript), compiling/running the generated code, and verifying the output against expected results.

Although Hanami is not designed to compete with commercial programming languages, this evaluation provided valuable insights into factors affecting language usability and transpiler correctness.

### B. Performance Considerations

The architecture was designed with performance considerations inspired by HPC compilation research [4], [7].

- **Modular Pipeline**: Allows potential parallelization of independent stages.
- **Memory Efficiency**: Use of appropriate data structures (e.g., AST, symbol tables).
- **Processing Time**: Algorithms chosen for reasonable performance (e.g., recursive descent parsing).

*1) Performance Results:* To evaluate the performance of the Garden language, we conducted an analysis of the execution time for each module during the compilation of two sample programs. The tests were designed to measure how the compiler performs with different types of programs.

*a) Test Environment:* The evaluations were performed in the following environment:

- IDE: Visual Studio Code v1.99.3
- CPU: Intel Core i7-11800H
- Measurement method: Using C++ `chrono` library

*b) Sample Programs:* Two different programs were used for the evaluation:

**Program 1: Simple Calculator**

Listing 7: Sample Program 1: Simple Calculator (Hanami-like syntax)

```
style <iostream>

garden SimpleProgram

species SimpleCalculator {
   open:
   grow add(int a, int b) -> int {
      blossom a + b;
   }

   grow subtract(int a, int b) -> int {
      blossom a - b;
   }

   grow multiply(int a, int b) -> int {
```

```
        blossom a * b;
    }

    grow divide(int a, int b) -> int {
        branch (b == 0) {
            blossom 0;
        }
        blossom a / b;
    }
};

grow mainGarden() -> int {
    SimpleCalculator calc;

    // Print results directly instead of assigning to
    variables
    bloom << "Addition: " << calc.add(5, 3) << "\n";
    bloom << "Subtraction: " << calc.subtract(10, 4) << "\n";
    bloom << "Multiplication: " << calc.multiply(6, 7) <<
    "\n";
    bloom << "Division: " << calc.divide(20, 5) << "\n";

    blossom 0;
}
```

**Program 2: Number Guessing Game**

Listing 8: Sample Program 2: Number Guessing Game (Hanami-like syntax)

```
style <iostream>
style <string>

garden Main

grow mainGarden() -> int
{
    std::string playerName;
    int guess;
    int secretNumber = 3;
    bool isCorrect;

    bloom << "Chao mung den voi tro choi doan so bi mat!" <<
    "\n";
    bloom << "Vui long nhap ten cua ban: ";

    water >> playerName;

    bloom << "Xin chao, " << playerName << "!" << "\n";
    bloom << "Hay doan mot so tu 1 den 5." << "\n";
    bloom << "Nhap so ban doan: ";

    water >> guess;

    branch (guess < 1)
    {
        bloom << "So ban nhap qua nho!" << "\n";
        bloom << playerName << ", so phai tu 1 den 5." <<
        "\n";
        bloom << "Tro choi ket thuc." << "\n";
        blossom 0;
    }

    branch (guess > 5)
    {
        bloom << "So ban nhap qua lon!" << "\n";
        bloom << playerName << ", so phai tu 1 den 5." <<
        "\n";
        bloom << "Tro choi ket thuc." << "\n";
        blossom 0;
    }

    isCorrect = (guess == secretNumber);

    branch (isCorrect)
    {
        bloom << "Chuc mung, " << playerName << "!" << "\n";
        bloom << "Ban da doan dung so bi mat: " <<
        secretNumber << "\n";
        bloom << "Ban that xuat sac!" << "\n";
    }

    branch (isCorrect == false)
    {
        bloom << "Rat tiec, " << playerName << "!" << "\n";
        bloom << "So ban doan khong dung." << "\n";
        bloom << "So bi mat la: " << secretNumber << "\n";
        bloom << "Hay thu lai lan sau nhe!" << "\n";
    }

    bloom << "Cam on ban da tham gia tro choi!" << "\n";

    blossom 0;
}
```

*c) Compilation Performance Results.:* The tables below present the execution time (in seconds) for each module during compilation of both programs:

TABLE IV: Execution times (seconds) for Simple Calculator program

| Run | Lexer | Parser | Sem. Anal. | Code Gen. | Total |
|---|---|---|---|---|---|
| 1 | 0.300513 | 0.196235 | 0.002480 | 0.005205 | 0.504433 |
| 2 | 0.269355 | 0.193547 | 0.001894 | 0.004807 | 0.469603 |
| 3 | 0.267332 | 0.243762 | 0.001938 | 0.005553 | 0.518585 |
| 4 | 0.339088 | 0.197705 | 0.001844 | 0.004578 | 0.543215 |
| 5 | 0.268074 | 0.215145 | 0.001955 | 0.015425 | 0.500599 |
| **Avg** | **0.289** | **0.209** | **0.002** | **0.007** | **0.507** |

TABLE V: Execution times (seconds) for Number Guessing Game program

| Run | Lexer | Parser | Sem. Anal. | Code Gen. | Total |
|---|---|---|---|---|---|
| 1 | 0.391050 | 0.310980 | 0.002120 | 0.005749 | 0.709899 |
| 2 | 0.423220 | 0.334670 | 0.002670 | 0.197931 | 0.958491 |
| 3 | 0.425052 | 0.335004 | 0.002520 | 0.006645 | 0.769221 |
| 4 | 0.410657 | 0.323180 | 0.002525 | 0.010270 | 0.746632 |
| 5 | 0.446285 | 0.292156 | 0.002157 | 0.006017 | 0.746615 |
| **Avg** | **0.419** | **0.319** | **0.002** | **0.045** | **0.786** |

*d) Comparative Analysis:* Comparing the performance metrics between the two programs:

- **Overall Performance**: Program 2 (Number Guessing Game) took approximately 55% longer to compile than Program 1 (Simple Calculator), with average total times of 0.786s vs. 0.507s.
- **Lexer**: The lexical analysis for Program 2 took 45% longer (0.419s vs. 0.289s), which correlates with its larger size and more complex structure.
- **Parser**: The parsing phase for Program 2 required about 53% more time (0.319s vs. 0.209s), reflecting the increased complexity of the control flow with multiple branch statements.
- **Semantic Analyzer**: The semantic analysis showed only a minimal increase (0.0024s vs. 0.0020s), suggesting that this phase scales well with program complexity.
- **Code Generator**: The most significant difference was in the code generation phase, where Program 2 required approximately 6.4 times longer (0.045s vs. 0.007s). This is largely due to the second run's anomalous value of 0.198s, which may indicate occasional optimization challenges or resource contention.

*e) Conclusions:* The performance analysis reveals that:

- The Lexer and Parser continue to be the most time-consuming phases of the compilation process for both programs, accounting for over 90% of the total time.
- The compilation time increases with program complexity and size, but not linearly across all phases.
- The Semantic Analyzer demonstrates excellent scalability, with minimal performance impact as program complexity increases.
- The Code Generator shows more variability in performance, suggesting it may benefit from further optimization to handle more complex programs consistently.
- For both programs, the total compilation time remains under one second, which is acceptable for development purposes, though further optimizations could improve the developer experience for larger codebases.

### C. Scalability Aspects

The architecture supports language and compiler evolution:

- **Feature Extensibility**: Clear separation of concerns facilitates adding new language features.
- **Target Platform Addition**: The code generator's visitor pattern makes adding new target languages relatively modular.
- **Optimization Expansion**: A framework for adding intermediate representations or optimization passes could be integrated later.

### D. Maintainability Factors

The architecture promotes maintainability:

- **Component Isolation**: Minimizing inter-component dependencies simplifies updates.
- **Interface Stability**: Clear data structures (Tokens, AST) serve as contracts between phases.
- **Testing Support**: Modular design facilitates unit testing of individual components.

### E. Quality Assurance

The architecture includes provisions for quality assurance:

- **Verification Points**: Semantic analysis acts as a key validation step.
- **Diagnostic Capabilities**: Emphasis on clear error reporting with source locations.
- **Tracing Infrastructure**: Potential for adding logging or debugging outputs throughout the pipeline.

### F. Limitations

As a student project, Hanami has several limitations that could be addressed in future work:

*1) Lack of Basic Data Structures:* The current version of Hanami lacks support for several essential data structures:

- **No array declaration syntax** - Cannot store and manipulate collections of data
- **Lack of element access operators** - Cannot access elements in data structures
- **No support for collections** - Limited ability to process sets and relational data

A desired implementation would include:

Listing 9: Desired array implementation example

```
// Array declaration
int numbers[5];
int initialized[] = {1, 2, 3, 4, 5};
// Array access
numbers[0] = 10;
bloom << numbers[2];
```

*2) Limitations in String Handling:* Hanami lacks comprehensive string manipulation features:

- No string interpolation
- Limited string operations
- No regular expression support

*3) Limited Control Structures:*

*a) Switch/Case Statements:* Hanami currently lacks a switch/case construct, which is essential for multi-path branching logic:

Listing 10: Desired switch/case example

```
switch (value) {
case 1:
   bloom << "One";
   break;
case 2:
   bloom << "Two";
   break;
default:
   bloom << "Other";
}
```

*b) Advanced Flow Control:* The language lacks several modern flow control constructs:

- Exception handling mechanisms
- Iterators and enhanced for-loops
- Pattern matching
- Asynchronous programming constructs

*4) Language Feature Limitations:*

*a) Object-Oriented Programming Limitations:* The current implementation has several OOP limitations:

- **Incomplete inheritance model** - Lacks multiple inheritance and interface implementation
- **No polymorphism** - Unable to use dynamic dispatch effectively
- **Limited access control** - Insufficient granularity in member visibility

*b) Functional Programming Limitations:* Hanami lacks modern functional programming features:

- No lambda expressions or first-class functions
- No higher-order functions
- No function composition utilities

These limitations also present learning opportunities for future iterations or subsequent projects in the CS3370 class.

### G. Future works

*1) Enhancing Language Syntax:*

*a) Data Structure Improvements:* To address the current lack of fundamental data structures, we will implement:

- **Comprehensive array system** with intuitive syntax for declaration, initialization, and access
- **Built-in collection types** including lists, sets, maps, and queues with optimized performance
- **Advanced data structure operations** including slicing, filtering, and mapping

Example of planned array implementation:

Listing 11: Planned array implementation example

```
// Flexible array declaration and initialization
arr[int] numbers = [1, 2, 3, 4, 5];
arr[string] names = ["Alice", "Bob", "Charlie"];
// Advanced operations
numbers[1..3] = [10, 20]; // Slice assignment
var doubled = numbers.map(n => n * 2); // Transformation
```

*b) String Handling Enhancement:* We will implement advanced string handling capabilities:

- **String interpolation** with embedded expressions
- **Built-in regular expression support** with intuitive syntax
- **Extensive string manipulation functions** for common operations

Example of planned string features:

Listing 12: Planned string features example

```
// String interpolation
var name = "World";
var greeting = "Hello ${name}!"; // "Hello World!"
// Regular expressions
var isEmail = "user@example.com" ~= /^[\w.]+@[\w.]+.\w+$/;
```

*2) Control Structure Enhancements:*

*a) Pattern Matching:* We will implement a powerful pattern matching system that goes beyond traditional switch/case:

Listing 13: Planned pattern matching example

```
match (value) {
case int n if n > 0 => bloom << "Positive number: ${n}";
case int n if n < 0 => bloom << "Negative number: ${n}";
case string s => bloom << "String: ${s}";
case Person(name: var n, age: var a) =>
   bloom << "Person ${n}, age ${a}";
case [var head, *var tail] =>
   bloom << "List with head ${head}";
case _ => bloom << "Default case";
}
```

*b) Exception Handling:* A robust exception handling system will be implemented:

Listing 14: Planned exception handling example

```
try {
   // Code that might throw exceptions
   var result = riskyOperation();
} catch (FileNotFound e) {
   // Handle specific exception
   bloom << "File not found: ${e.message}";
} catch (e: NetworkError | TimeoutError) {
   // Handle multiple exception types
   bloom << "Connection issue: ${e.message}";
} finally {
   // Always executed
   cleanup();
}
```

*3) Unique Features and Advantages:*

*a) Bidirectional Type Inference:* Unlike many languages with limited type inference, Hanami will feature bidirectional type inference that provides both safety and conciseness:

- **Local and global inference** that reduces type annotations while maintaining type safety
- **Partial type annotations** allowing developers to specify only critical types
- **Flow-sensitive typing** that understands type changes through control flow

Listing 15: Planned type inference example

```
// Type inferred from initialization
var x = 5; // inferred as int
var y = "hello"; // inferred as string
// Flow-sensitive typing
var z = getValueFromNetwork();
if (z is int) {
   // z is treated as int in this scope
   bloom << z + 10;
}
```

*4) Context-Aware Compilation:* Hanami will introduce a context-aware compilation system building on effect handler research [14], [16], [17].

- **Adaptive optimization** based on usage patterns
- **Contextual code suggestions** during development
- **Runtime performance profiling** that feeds back into the compiler

This system will allow Hanami to optimize code based on how it's actually used rather than generic optimization strategies.

## V. Interactive Language Playground: Development and Deployment

As the Hanami language advanced through its implementation stages, a critical question emerged: how could we provide an accessible way for users to interact with and explore the language without requiring complex local setup? Traditional compiler projects often involve significant technical overhead for end-users, potentially limiting engagement with the language itself. To address this challenge, we conceptualized and developed an interactive web-based playground that would showcase Hanami's capabilities while providing valuable insights into the compilation process.

### A. Conceptualization and Motivation

The playground system was born from several educational objectives:

- **Accessibility**: Eliminating installation barriers by providing immediate access to Hanami through a web browser.
- **Transparency**: Exposing the inner workings of the compiler pipeline to facilitate understanding of language processing stages.
- **Interactivity**: Allowing real-time experimentation with code snippets and immediate feedback.
- **Educational Value**: Demonstrating the transformation from source code to intermediate representations to final output.

Traditional compiler development typically separates the language implementation from user interfaces. However, by integrating a modern web interface, we sought to create an educational tool that would bridge theoretical compiler concepts with practical language exploration, similar to web tools used in programming education [4], [12].

### B. System Architecture

The playground was designed as a distributed system with two primary components:

- **Frontend Interface**: A Next.js application providing an intuitive code editor with syntax highlighting, module selection, and output visualization.
- **Backend API**: An Express.js server connecting the frontend to the C++ compiler modules through a RESTful interface.

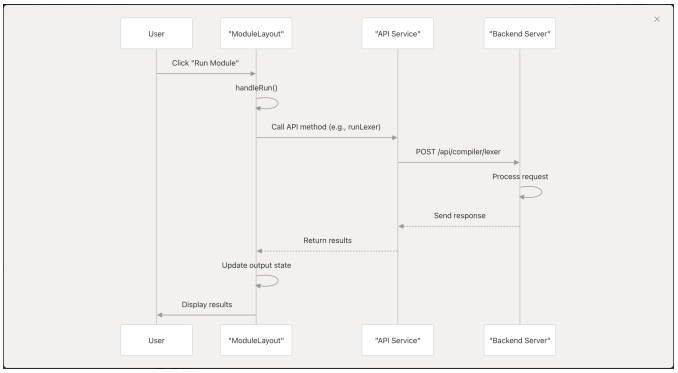



Fig. 6: Hanami Frontend Playground Activity Diagram

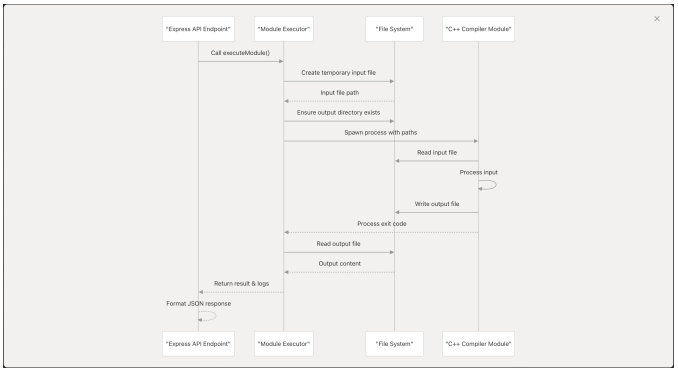



Fig. 7: Hanami Backend Playground Activity Diagram

*1) Frontend Implementation:* The frontend was developed using Next.js, a React framework enabling server-side rendering and static site generation. Key components included:

- **Monaco Editor Integration**: We implemented the Monaco code editor with custom Hanami language definition, providing syntax highlighting and auto-completion features specifically tailored to Hanami's nature-inspired syntax.
- **xterm.js Terminal**: An interactive terminal component displays compilation logs and process information, replicating the experience of using a compiler from the command line.
- **Module-Specific Pages**: Dedicated interfaces for each compiler stage (lexer, parser, semantic analyzer, code generator) with appropriate visualizations for their outputs.

The frontend architecture followed a component-based approach with shared layouts and context providers for state management. A responsive design ensured usability across different device types and screen sizes.

*2) Backend Implementation:* The backend API was built with Express.js and TypeScript, focusing on:

- **RESTful Endpoints**: Providing clear API endpoints for each compiler module (`/api/compiler/lexer`, `/api/compiler/parser`, etc.).
- **Module Integration**: Executing the C++ compiler modules through child processes, with robust error handling and logging.
- **File Management**: Implementing temporary file handling for inputs and outputs during compilation stages.
- **Cross-Origin Resource Sharing**: Configuring appropriate CORS headers to enable secure communication between the frontend and backend.

The API design allowed for both direct code submissions and the passing of intermediate representations between compilation stages, providing flexibility in how users interact with the compiler pipeline.

### C. Inter-Module Communication

One of the most educational aspects of the playground is its transparent handling of intermediate representations:

Listing 16: Sample Communication Between Frontend and Backend API

```
// Frontend request to the lexer endpoint
async function processLexerModule(code) {
  const response = await
  fetch(`${API_URL}/compiler/lexer`, {
    method: 'POST',
    headers: { 'Content-Type':
    'application/json' },
    body: JSON.stringify({ code })
  });

  const result = await response.json();

  // Display tokens and prepare for potential
  parser stage
  displayTokens(result.output);
  return result;
}
```

This approach enabled users to observe how source code transforms through each compilation stage, from lexical tokens to abstract syntax trees, semantic representations, and finally to generated code in target languages.

### D. Deployment Strategy

To ensure accessibility, we deployed the playground using cloud platforms optimized for their respective components:

- **Frontend Deployment**: The Next.js application was deployed to Netlify, leveraging its continuous deployment pipeline and global CDN for optimal performance.
- **Backend Deployment**: The Express API and C++ compiler modules were deployed to Railway, which provided the necessary computational resources for running compilation processes.

The deployment configuration was managed through declaration files (`netlify.toml` and `railway.toml`) and deployment scripts that automated the build and deployment process:

Listing 17: Backend Deployment Configuration (railway.toml)

```
[build]
  builder = "nixpacks"
  buildCommand = "./download-dependencies.sh
  && ./build-backend.sh && cd
  development/MODULES && make"

[deploy]
  startCommand = "cd /app/development/backend
  && node dist/index.js"
  healthcheckPath = "/api/compiler/logs"
  healthcheckTimeout = 100
  restartPolicyType = "on_failure"

[env]
  PORT = "8080"
  NODE_ENV = "production"
  NPM_CONFIG_LEGACY_PEER_DEPS = "true"
  NIXPACKS_NODE_VERSION = "18"
  PUBLIC_URL =
  "https://hanami-backend-production.up.railway.app"
  MODULES_PATH = "/app/development/MODULES"
```

This deployment strategy ensured that the playground remained consistently available to users without requiring maintenance of dedicated infrastructure.

### E. Educational Impact

The playground has proven to be a valuable educational tool, offering several benefits:

- **Immediate Engagement**: New users can begin experimenting with Hanami immediately without installation.
- **Pipeline Visualization**: The step-by-step processing helps users understand compiler concepts that are typically abstract.
- **Debugging Assistance**: Visible intermediate representations simplify the debugging of syntax and semantic errors.
- **Research Demonstration**: The playground serves as a demonstration platform for presenting Hanami's design concepts.

By integrating modern web technologies with traditional compiler development, the playground act as an interactive learning zone with immediate feedback for anyone with interest in Hanami.

### F. Online Resources and Live Demonstration

To promote access to the Hanami language and encourage further exploration, we have made all resources publicly available:

- **Source Code Repository**: The complete source code, including both the compiler modules and the playground interface, is available on GitHub at https://github.com/kiyo9w/Hanami-CS370.
- **Live Frontend Demo**: The interactive playground frontend is deployed at https://hanami-fe.netlify.app/, providing immediate access to the Hanami language exploration interface.
- **Backend API**: The compiler backend services are accessible at https://hanami-backend-production.up.railway.app/, which hosts the RESTful API endpoints.

The following figures demonstrate the interactive playground interface:

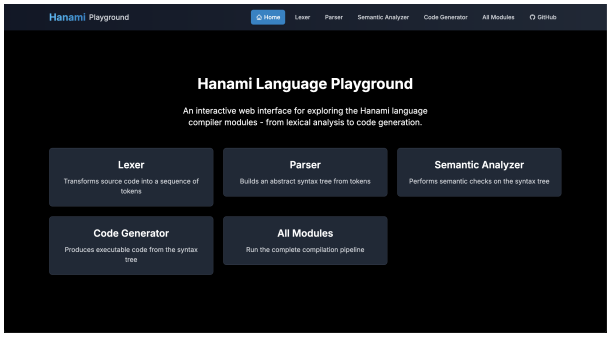



Fig. 8: Hanami Playground Homepage with Module Selection

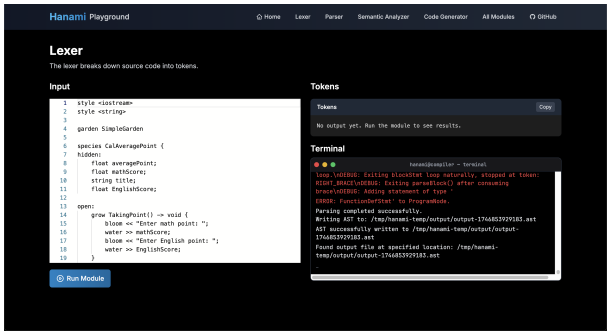



Fig. 9: Lexical Analysis Module Interface with Sample Code

## VI. Conclusion

This paper presented Hanami, a programming language developed as an educational project within the CS3370 course. By designing and implementing a transpiler targeting multiple languages (C++, Java, Python, JavaScript), we gained hands-on experience with lexical analysis, parsing, semantic analysis, and code generation. The nature-inspired syntax and features like multi-target transpilation served as practical vehicles for exploring language design principles and trade-offs. The modular architecture proved effective for managing complexity

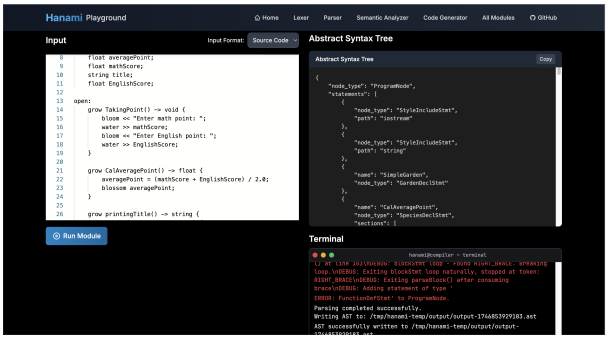



Fig. 10: Parser Module Interface with Abstract Syntax Tree Output and Terminal Execution Logs

and demonstrated key software engineering concepts relevant to compiler construction. The Hanami project successfully met its educational objectives, demonstrating principles of transpiler design [3], [6], [18] and language theory [10], [15], [19].

As we conclude this project, we recognize that Hanami's greatest value lies not in its potential for commercial adoption but in its demonstration of how programming language theory can be made tangible through thoughtful implementation. The interactive playground, compiler architecture, and transpilation approach collectively showcase the educational value of language implementation as a learning bridge for computer science education.

## Acknowledgements

We would like to express our deepest gratitude to Dr. Nguyen Dinh Han for his invaluable lectures that guided us in the development of this project. His expertise in programming language theory and compiler design provided the foundation for our work on Hanami.

## References

[1] K. C. Louden and K. A. Lambert, *Programming Languages: Principles and Practices*, 3rd ed. Cengage Learning, 2011.

[2] S. Bhatia, S. Kohli, S. A. Seshia, and A. Cheung, "Building code transpilers for domain-specific languages using program synthesis," in *Proceedings of the European Conference on Object-Oriented Programming*, vol. 263, 2023, pp. 1–30.

[3] Rohit, A. Kumar, and S. Singh, "Phases of a transpiler," *International Journal of Computer Applications*, vol. 127, no. 1, pp. 20–25, 2015.

[4] M. Li, Y. Zhao, L. Luo, Y. Shen, D. Peng, Z. Xie, and J. Liu, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2020.

[5] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[6] K. Thompson and J. Wilson, "Source-to-source compiler - an overview," in *Handbook of Programming Languages*. ScienceDirect, 2023, pp. 123–156.

[7] R. Milewicz, P. Pirkelbauer, P. Soundararajan, H. Ahmed, and T. Skjellum, "Source-to-source compilers in hpc: A literature review," in *Proceedings of the International Conference on High Performance Computing*, 2020, pp. 123–135.

[8] J. G. Siek and W. Taha, "Gradual typing for functional languages," *Scheme and Functional Programming Workshop*, pp. 81–92, 2006.

[9] K. D. Lee, *Foundations of Programming Languages*, 2nd ed. Springer, 2017.

[10] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.

[12] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," *EPFL Technical Report*, no. IC/2004/64, 2004.

[13] G. Plotkin and M. Pretnar, "A logic for algebraic effects," *Logic in Computer Science*, pp. 118–129, 2009.

[14] S. Lindley, M. Pretnar, and P. Schuster, "Effect handlers and general purpose languages," Shonan Meeting Reports, Tech. Rep. 146, 2023.

[15] M. Felleisen, "On the expressive power of programming languages," *Science of Computer Programming*, vol. 17, no. 1-3, pp. 35–75, 1991.

[16] M. Pretnar, "An introduction to algebraic effects and handlers," *Electronic Notes in Theoretical Computer Science*, vol. 319, pp. 19–35, 2015.

[17] J. I. Brachthäuser, P. Schuster, and K. Ostermann, "Effect handlers for the masses," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[18] G. Castagna, V. Lanvin, T. Petrucciani, and J. G. Siek, "Gradual typing: A new perspective," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–32, 2019.

[19] D. Grossman, "Typed memory management in a calculus of capabilities," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003, pp. 262–275.

# VII. Appendix

## A. Contribution Table

| ID | Member | Task Weight | Details of Contribution |
|---|---|---|---|
| 1. | Dat | 25% | - Implemented compiler basics documentation, including the Markdown document for the compiler stages.<br>- Improved lexer functionality for additional tokens, comments, and error handling.<br>- Finalized report introduction and evaluation discussion sections. |
| 2. | An | 20% | - Created a visual pipeline diagram outlining the source code journey to executable.<br>- Designed and implemented the recursive descent parser.<br>- Conducted unit testing for lexer and parser components with test cases. |
| 3. | Duong | 20% | - Developed a simple arithmetic lexer prototype.<br>- Implemented a semantic analyzer for type checking and AST context enrichment.<br>- Designed detailed system architecture and created demonstrative code examples. |
| 4. | Trung | 35% | - Coordinated and documented team meetings for core compiler concepts.<br>- Organize in-person code review session, create detailed documentation prototype for modules to guide development.<br>- Created the code generation module and integrated error reporting mechanisms.<br>- Created the Makefile and input/output system to link & coordinate the compiler build and run structure.<br>- Created an web application for integrated program playground.<br>- Finalized the complete project report, ensuring consistency in documentation. |

TABLE VI: Task Distribution Table

*B. Progression By Sprints*

| Sprii | Weel | Member | Task | Description | Deadline | Deliverable |
|---|---|---|---|---|---|---|
| 1 | 1 | Dat | Compiler Basics Documentation | Create a simplified Markdown document explaining core compiler stages | Mar 8, 2025 | Document explaining compiler pipeline |
| | | An | Visual Pipeline Diagram | Create diagram illustrating journey from source code to executable | Mar 8, 2025 | Diagram highlighting key compiler phases |
| | 2 | Duong | Simple Lexer Exercise | Implement basic C++ program tokenizing simple arithmetic expression | Mar 15, 2025 | Working tokenizer for arithmetic expressions |
| | | Trung | Design Overview | Write overview discussing differences between compiled and interpreted languages | Mar 15, 2025 | Document on language design impact |
| | | | Meetings Organization | Organize meetings to discuss basic compiler concepts | Weeks 1-2 | Coordinated team discussions |
| | | | Week Review & Feedback | Collect feedback, compile brief review of learning | Mar 15, 2025 | Review summary with planning adjustments |
| 2 | 1 | Dat | Lexer Enhancements | Begin developing C++ based lexer with additional tokens | Mar 22, 2025 | Initial lexer implementation |
| | | An | Parser Framework | Develop basic parser framework | Mar 22, 2025 | Initial parser structure |
| | 2 | Duong | Semantic Analyzer Module | Implement C++ semantic analysis module for type checking, symbol tables, IR generation | Mar 29, 2025 | Semantic analyzer enriching AST with context |
| | | Trung | Code Generation Module | Create module converting IR into C++ code for transpiling | Mar 29, 2025 | Working transpiler producing C++ code |
| | | | Integration & Error Reporting | Integrate code generation with other components | Mar 29, 2025 | Seamless module with clear error reporting |
| 3 | 1 | Trung | Module Integration (Initial) | Begin linking of compiler modules, define intermediate formats | Apr 5, 2025 | Initial integration framework |
| | | | Report Structure | Finalize report structure | Apr 5, 2025 | Report outline document |
| | | Dat | Module Debugging (Initial) | Begin debugging efforts based on initial tests | Apr 5, 2025 | First round bug fixes |
| | 2 | Trung | Integration Continued | Continue integration efforts, focus on resolving issues | Apr 12, 2025 | Partially integrated pipeline |
| | | Dat | Module Debugging & Extension | Continue debugging efforts and extend modules | Apr 12, 2025 | More robust compiler modules |
| | | An | Unit Testing (Lexer & Parser) | Develop/execute min 5 tests each for Lexer and Parser | Apr 12, 2025 | Verified Lexer and Parser modules |
| | | Duong | Unit Testing (Semantic Analyzer & Code Gen) | Develop/execute min 5 tests each | Apr 12, 2025 | Verified semantic/code gen modules |
| | 3 | Trung | Playground application - Phase 1 | Initiate Playground web application with basic syntax highlighting IDE and working compiler modules | Apr 19, 2025 | Basic Web Application |
| | | Dat | AST -> IR Refinement | Implement changes for improved IR | Apr 19, 2025 | Distinct IR representation |
| | | | Documentation: Introduction | Draft Introduction and Related Work sections | Apr 19, 2025 | Comprehensive introduction draft |
| | | An | Documentation: Overview | Draft Overview section using Hanami Syntaxes.docx | Apr 19, 2025 | Clear language overview draft |
| | | | Visualization: Sequence Diagram | Create Sequence Diagram for key scenario | Apr 19, 2025 | Visual explanation of component interaction |
| | | Duong | Documentation: System Architecture | Draft System Architecture section | Apr 19, 2025 | Detailed internal design draft |
| | | | Visualization: UML Class Diagram | Create UML Class Diagram for key data structures | Apr 19, 2025 | Visual representation of data structures |

TABLE VII – Continued from previous page

| Sprii | Weel | Member | Task | Description | Deadline | Deliverable |
|---|---|---|---|---|---|---|
| 4 | 1 | Trung | Code Implementation Finishing Touches | Finalize core language features; perform code cleanup | Apr 26, 2025 | Stable, feature-complete codebase |
| | | | Language Design Details | Draft detailed language design section | Apr 26, 2025 | Comprehensive language design draft |
| | | An | Overview Finalization | Refine and finalize language overview section | Apr 26, 2025 | Finalized overview section |
| | | Duong | Limitations and Future Work | Draft limitations and future work section | Apr 26, 2025 | Draft of limitations/future work |
| | | | Code Examples for Report | Prepare 1-2 clear code examples demonstrating features | Apr 26, 2025 | Polished code examples |
| | 2 | Trung | Abstract & Conclusion | Draft abstract and conclusion sections | May 3, 2025 | Complete abstract and conclusion |
| | | | Playground application - Phase 2 | Hosting - deploy and host Playground Frontend and Backend project on railway and netlify | May 10, 2025 | Accessible online endpoints for both FE&BE |
| | | | Final Report Assembly | Consolidate all sections, ensure consistency, proofread | May 3, 2025 | Complete, polished final report |
| | | Dat | Introduction & Related Work Finalization | Refine and finalize these sections | May 3, 2025 | Finalized introduction and related work |
| | | | Evaluation & Discussion | Draft evaluation and discussion sections | May 3, 2025 | Complete evaluation and discussion |
| | 3 | Trung | Playground application - Phase 3 | Implement enhancements (error highlighting, improved snippets) | May 10, 2025 | Improved Web Application |
| | | An | References | Compile and format references list | May 10, 2025 | Complete references section |
| | | | Diagram Review (Sequence) | Review sequence diagram for accuracy and integration | May 10, 2025 | Reviewed sequence diagram |
| | | Duong | System Architecture Finalization | Refine and finalize system architecture section | May 10, 2025 | Finalized system architecture |
| | | | Diagram Review (UML) | Review UML class diagram for accuracy and integration | May 10, 2025 | Reviewed UML class diagram |
| | | Dat | Code Comments & Readability | Review core compiler logic modules, improve comments | May 10, 2025 | Improved code clarity |

# A Hanami Language Reference

## A.1 Keyword Mapping

Table 1: Core Hanami keywords and their C++ equivalents

| Hanami | C++ Equivalent | Meaning |
|---|---|---|
| `garden <Name>` | `namespace <Name>...` | Declares a namespace |
| `species <Name>` | `class <Name>...` | Declares a class |
| `open:` | `public:` | Public section specifier |
| `hidden:` | `private:` | Private section specifier |
| `grow <f>(...) -> <t>` | `<t> <f>(...)` | Declares a function |
| `bloom « x;` | `std::cout « x;` | Console output |
| `water » x;` | `std::cin » x;` | Console input |
| `branch (cond)...` | `if (cond)...` | If statement |
| `else branch (cond)` | `else if (cond)` | Else-if clause |
| `else ...` | `else ...` | Else clause |

# B UML Diagrams

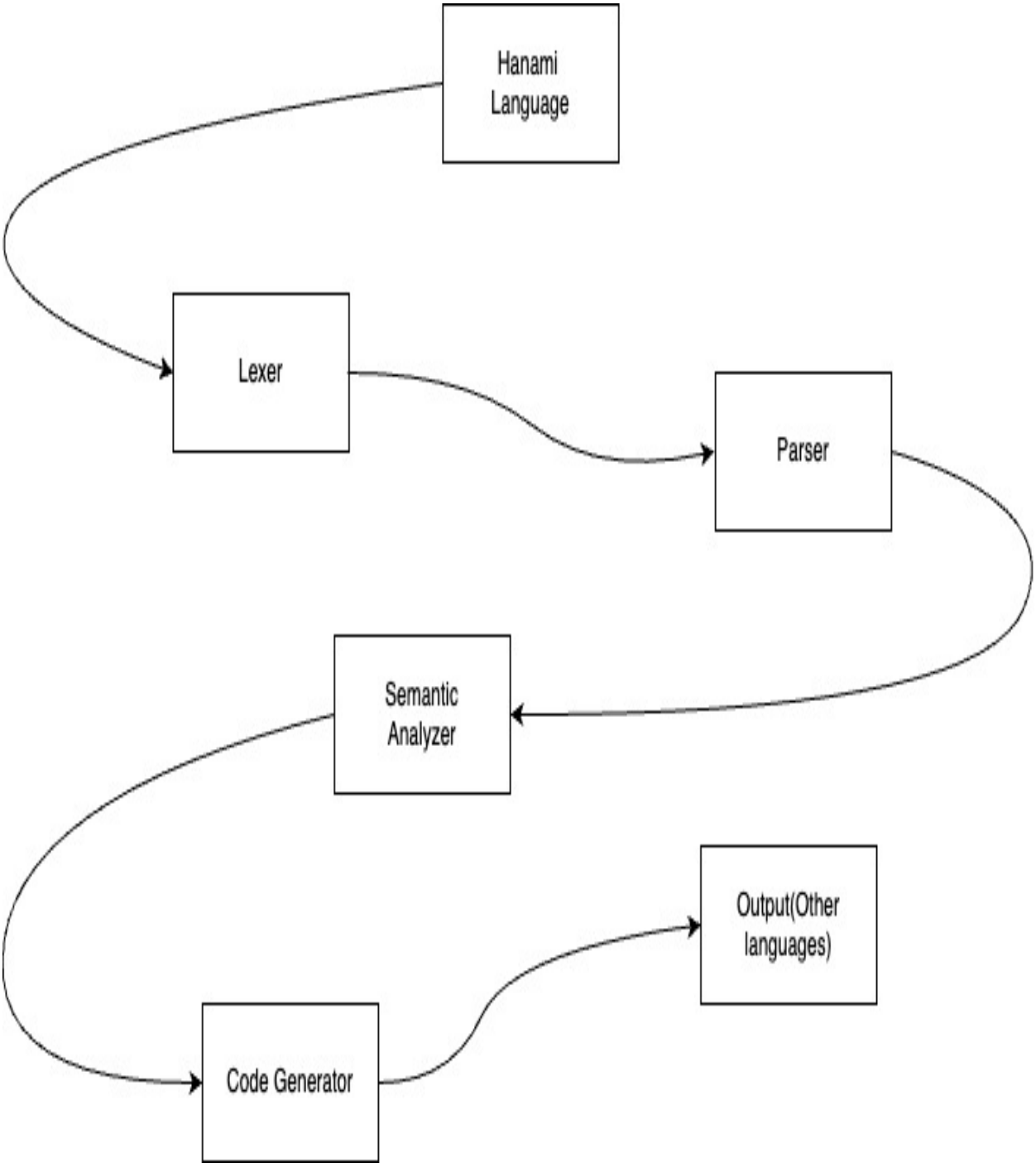
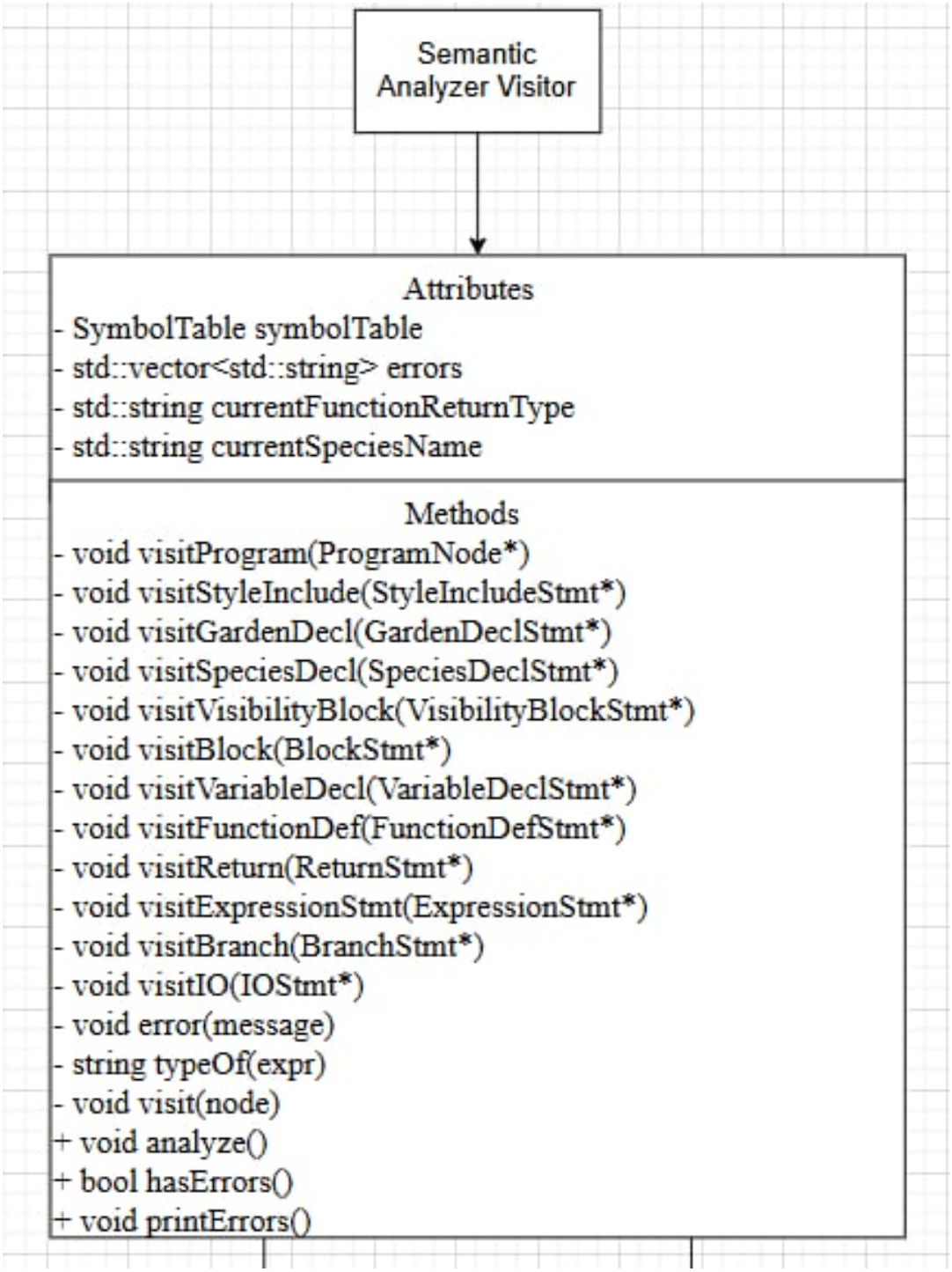
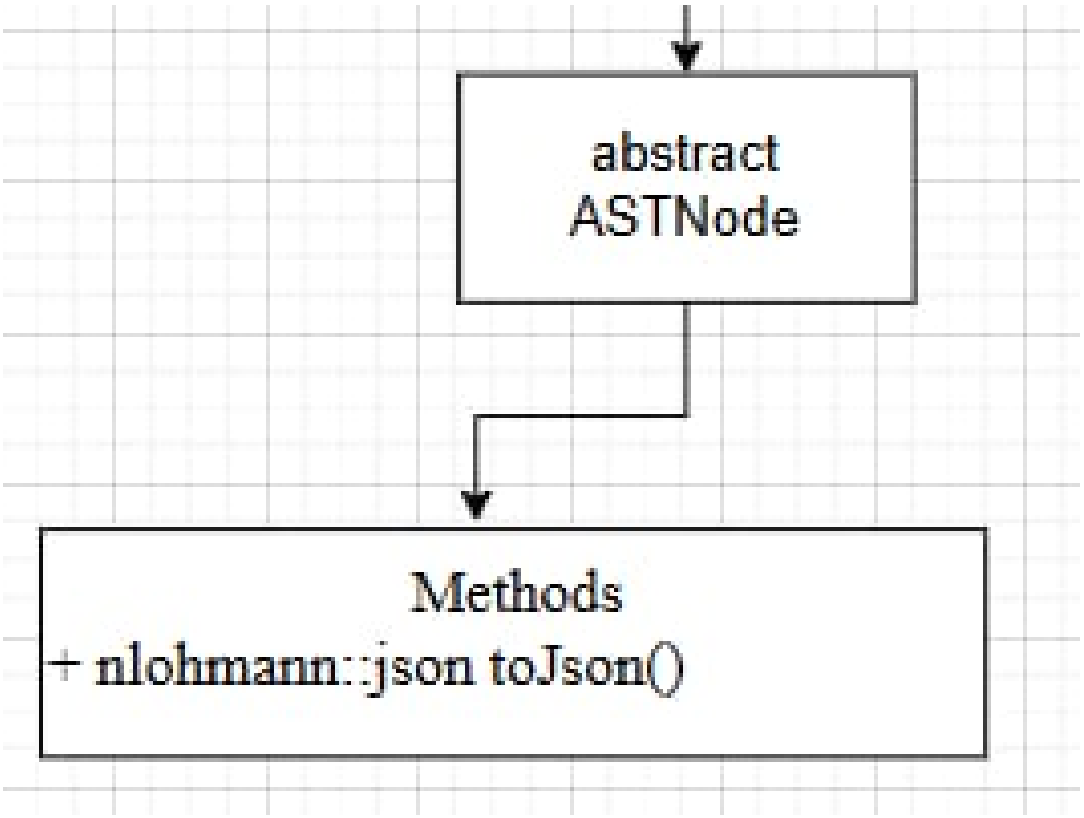
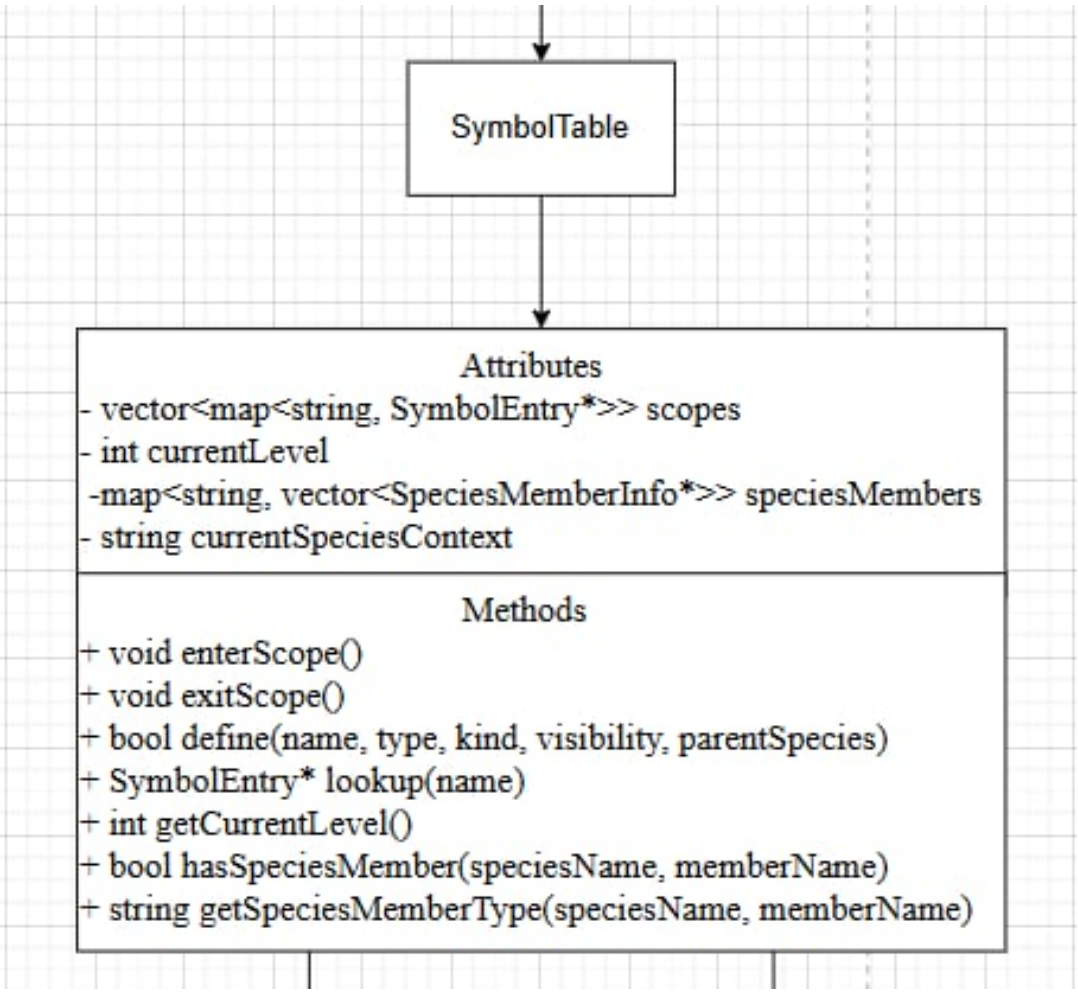



Figure 1: High-level flow from Hanami Language through Lexer, Parser, Semantic Analyzer, to Code Generator and target outputs.

# C Semantic Analyzer Visitor


Semantic Analyzer Visitor
Attributes
- SymbolTable symbolTable
- std::vector<std::string> errors
- std::string currentFunctionReturnType
- std::string currentSpeciesName
Methods
- void visitProgram(ProgramNode*)
- void visitStyleInclude(StyleIncludeStmt*)
- void visitGardenDecl(GardenDeclStmt*)
- void visitSpeciesDecl(SpeciesDeclStmt*)
- void visitVisibilityBlock(VisibilityBlockStmt*)
- void visitBlock(BlockStmt*)
- void visitVariableDecl(VariableDeclStmt*)
- void visitFunctionDef(FunctionDefStmt*)
- void visitReturn(ReturnStmt*)
- void visitExpressionStmt(ExpressionStmt*)
- void visitBranch(BranchStmt*)
- void visitIO(IOStmt*)
- void error(message)
- string typeOf(expr)
- void visit(node)
+ void analyze()
+ bool hasErrors()
+ void printErrors()


abstract
ASTNode
Methods
+ nlohmann::json toJson()


SymbolTable
Attributes
- vector<map<string, SymbolEntry*>> scopes
- int currentLevel
-map<string, vector<SpeciesMemberInfo*>> speciesMembers
- string currentSpeciesContext
Methods
+ void enterScope()
+ void exitScope()
+ bool define(name, type, kind, visibility, parentSpecies)
+ SymbolEntry* lookup(name)
+ int getCurrentLevel()
+ bool hasSpeciesMember(speciesName, memberName)
+ string getSpeciesMemberType(speciesName, memberName)

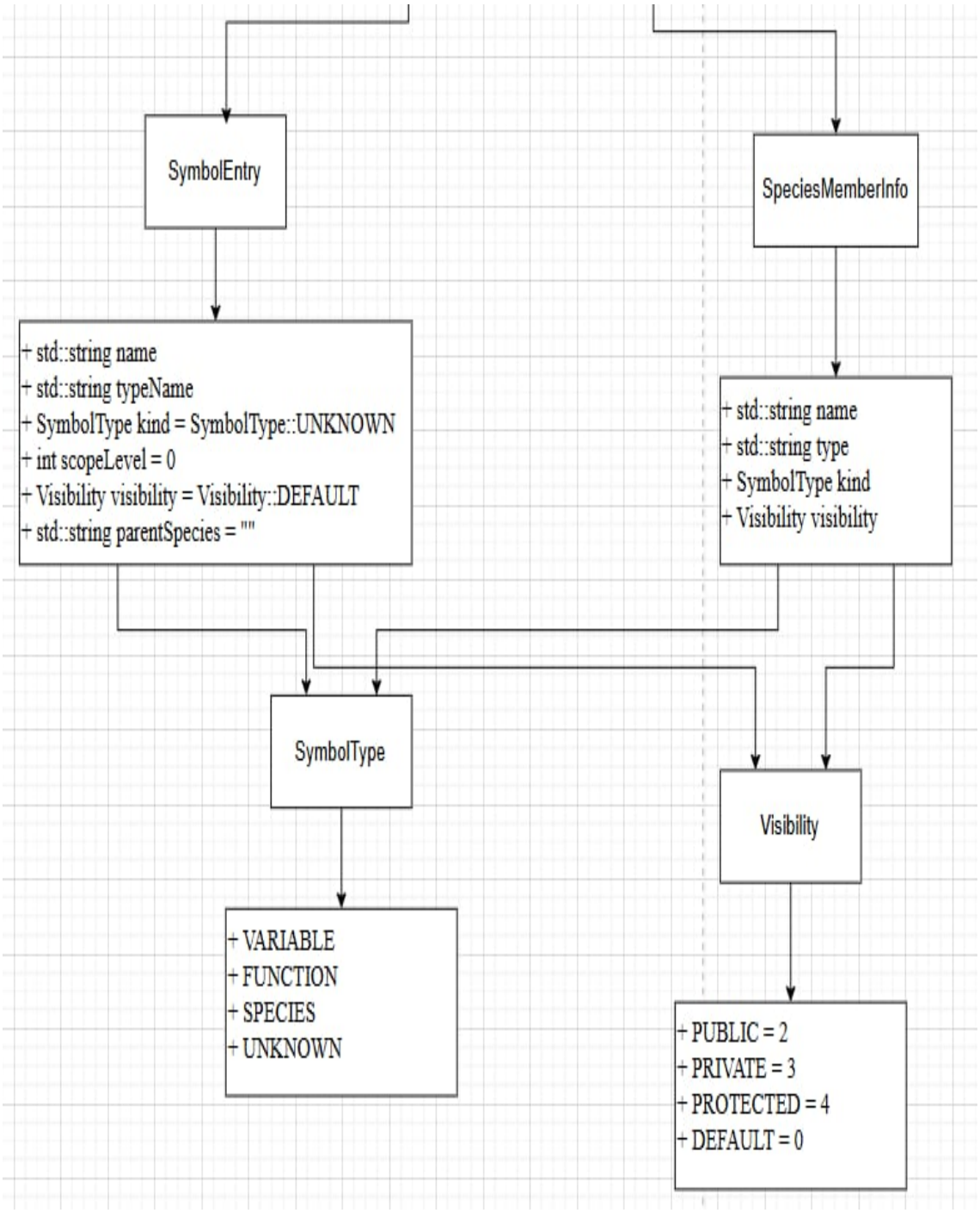



Figure 2: Class diagram of the Semantic Analyzer visitor, symbol table, and related entries.

# D Parser Structure (Part 1)

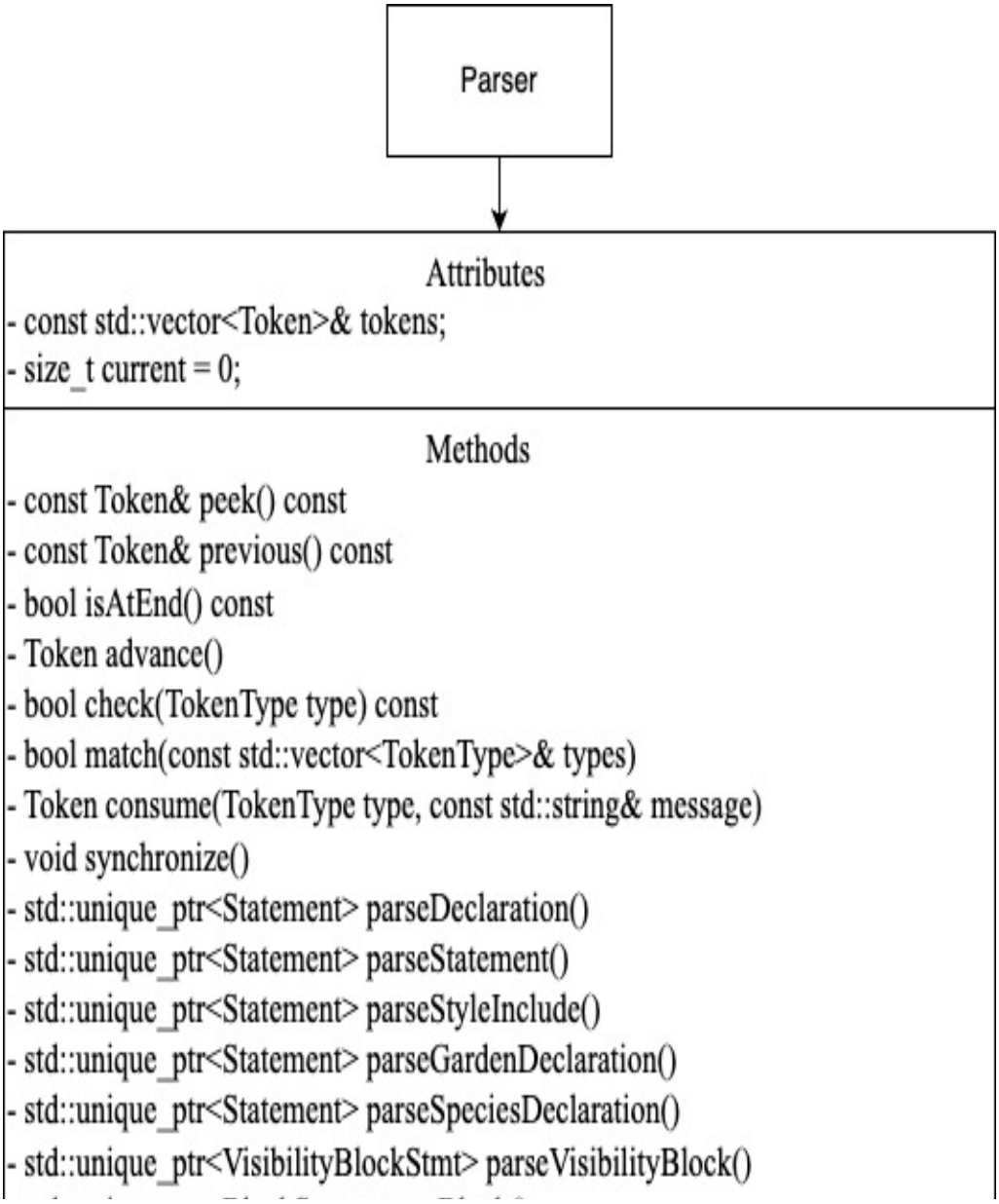



Figure 3: Parser class attributes and parsing methods (overview, part 1).

# E Parser Structure (Part 2)

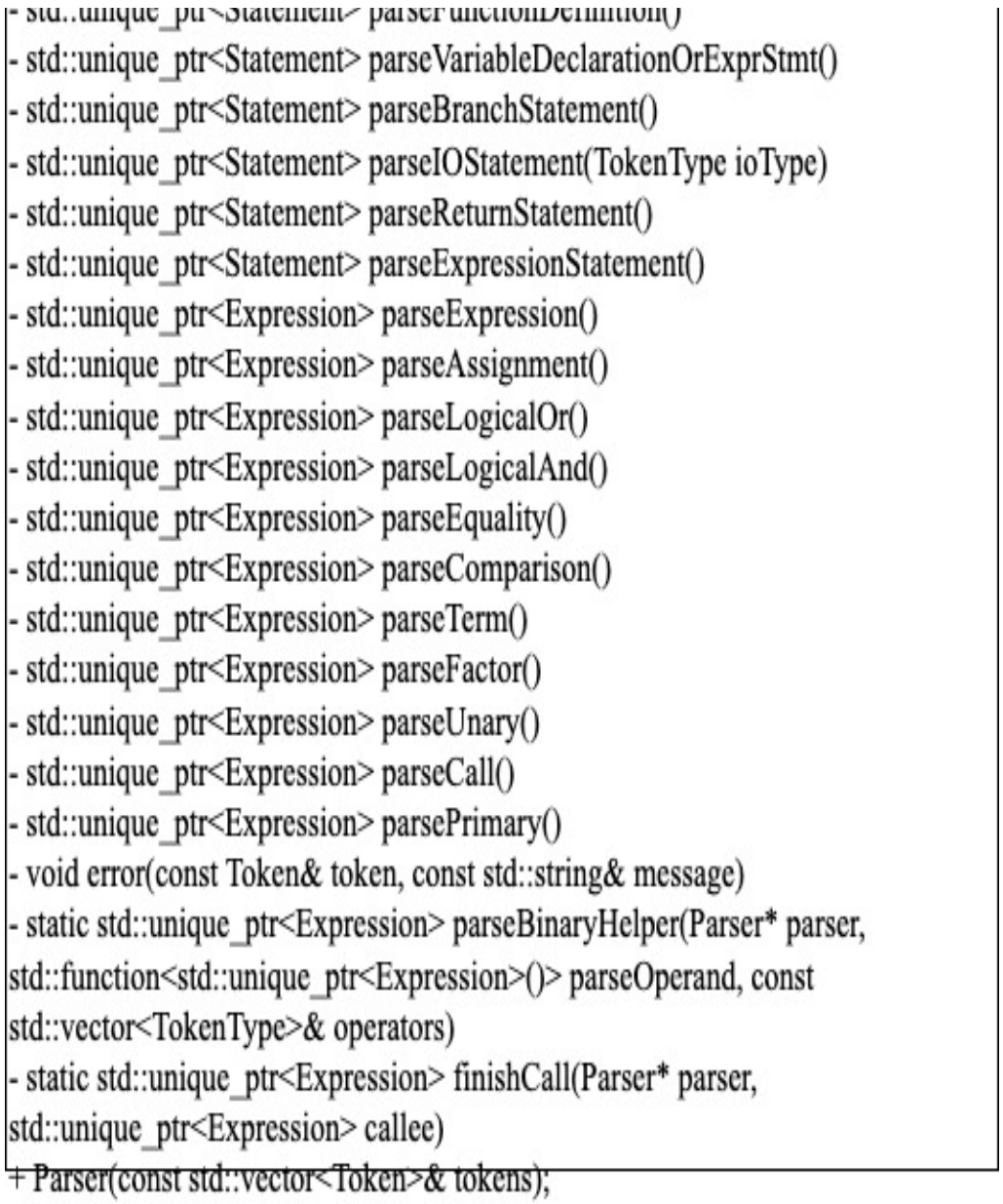
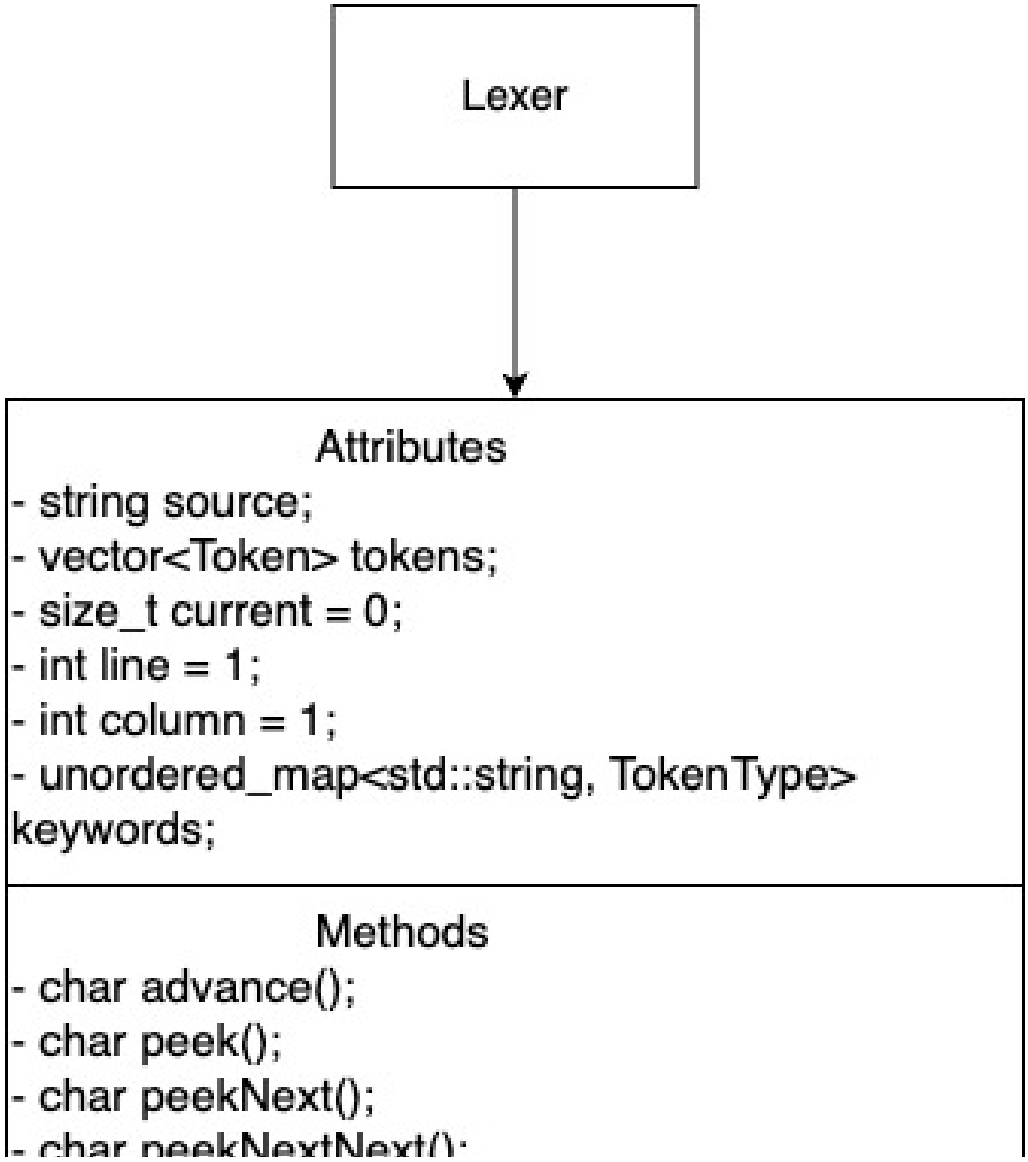
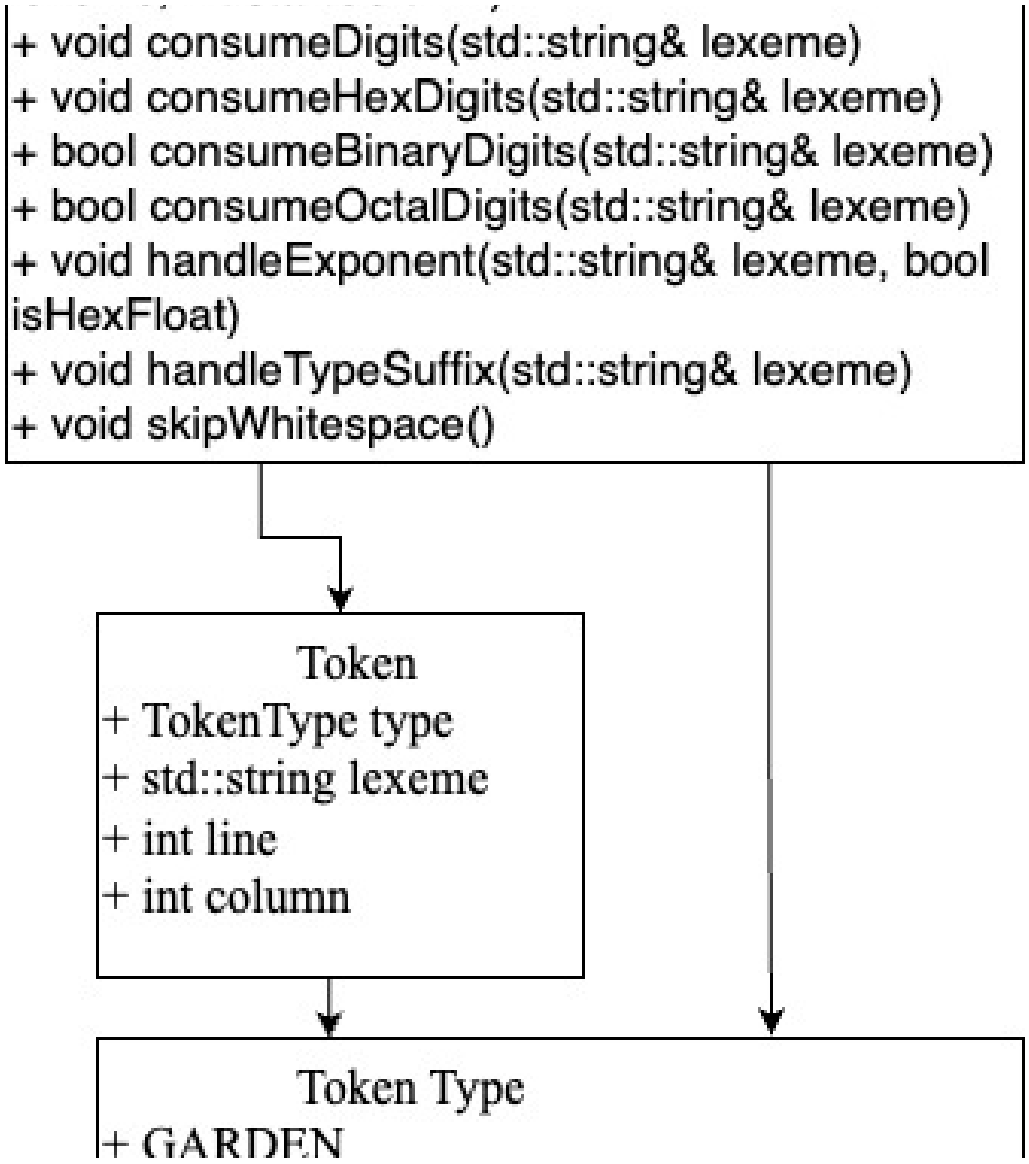


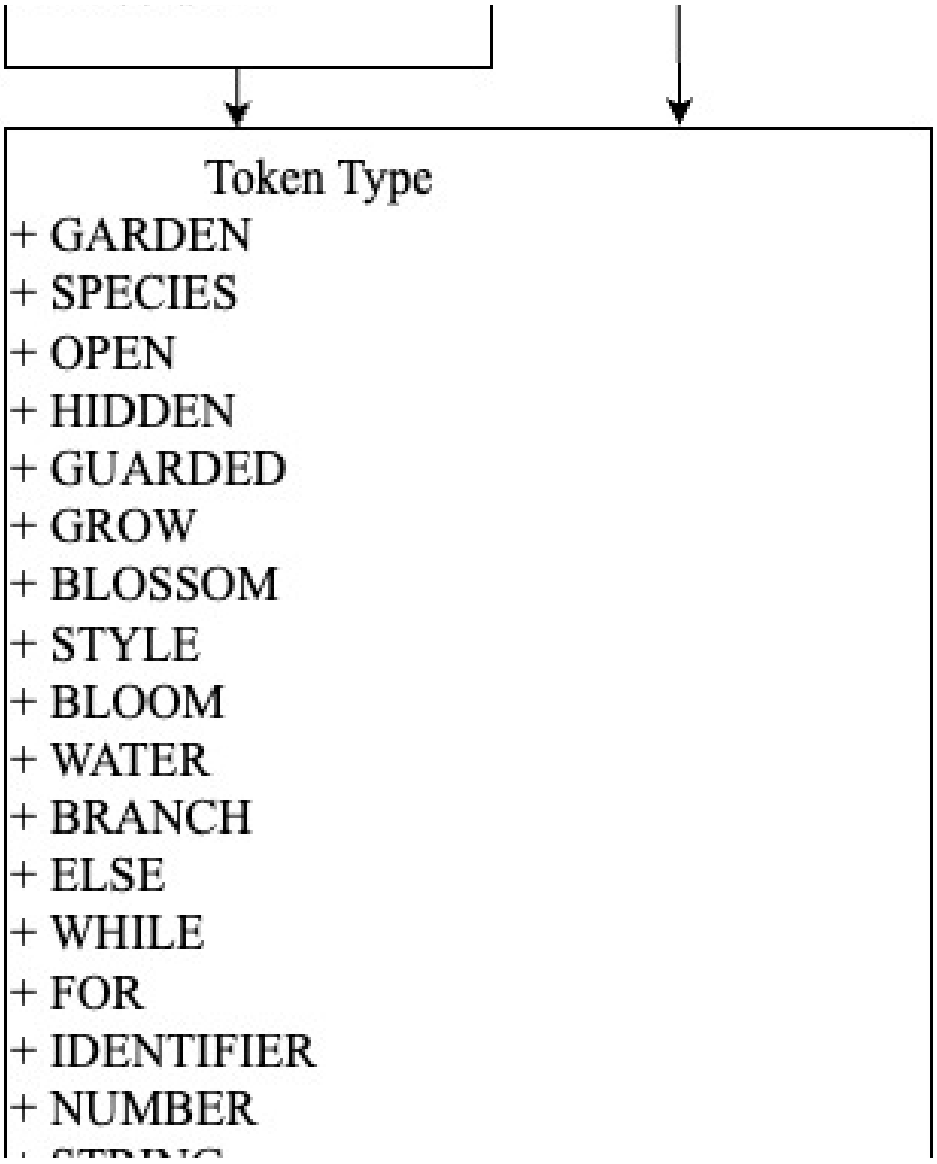

Figure 4: Parser class attributes and parsing methods (details, part 2).

## F Lexer Implementation



| Methods |
| --- |
| - char advance();<br>- char peek();<br>- char peekNext();<br>- char peekNextNext();<br>- bool isEnd();<br>- bool match(char expected);<br>+ Lexer(const std::string& source)<br>+ std::vector<Token> scanTokens()<br>+ Token scanToken()<br>+ void initKeywords()<br>+ Token identifier()<br>+ Token Number()<br>+ Token string()<br>+ Token skipComment()<br>+ Token handleSpecialNumber(std::string& lexeme, int startColumn)<br>+ void consumeDigits(std::string& lexeme)<br>+ void consumeHexDigits(std::string& lexeme)<br>+ bool consumeBinaryDigits(std::string& lexeme)<br>+ bool consumeOctalDigits(std::string& lexeme) |


+ void consumeDigits(std::string& lexeme)
+ void consumeHexDigits(std::string& lexeme)
+ bool consumeBinaryDigits(std::string& lexeme)
+ bool consumeOctalDigits(std::string& lexeme)
+ void handleExponent(std::string& lexeme, bool isHexFloat)
+ void handleTypeSuffix(std::string& lexeme)
+ void skipWhitespace()
Token
+ TokenType type
+ std::string lexeme
+ int line
+ int column
Token Type
+ GARDEN


Token Type
+ GARDEN
+ SPECIES
+ OPEN
+ HIDDEN
+ GUARDED
+ GROW
+ BLOSSOM
+ STYLE
+ BLOOM
+ WATER
+ BRANCH
+ ELSE
+ WHILE
+ FOR
+ IDENTIFIER
+ NUMBER

+ WHILE
+ FOR
+ IDENTIFIER
+ NUMBER
+ STRING
+ TRUE
+ FALSE
+ PLUS
+ MINUS
+ STAR
+ SLASH
+ ASSIGN
+ EQUAL
+ NOT_EQUAL
+ LESS
+ LESS_EQUAL
+ GREATER
+ GREATER_EQUAL
+ AND
+ OR

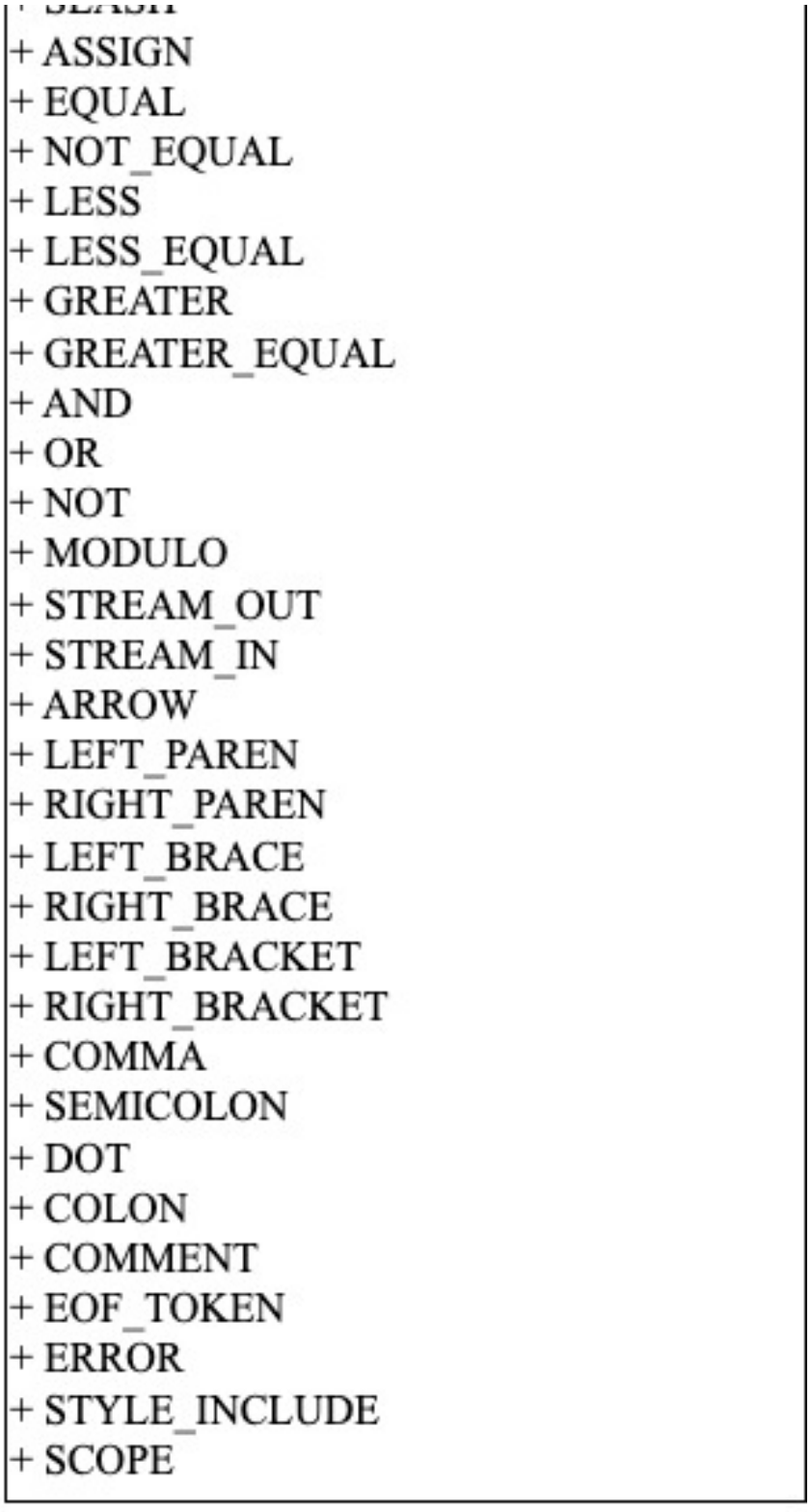
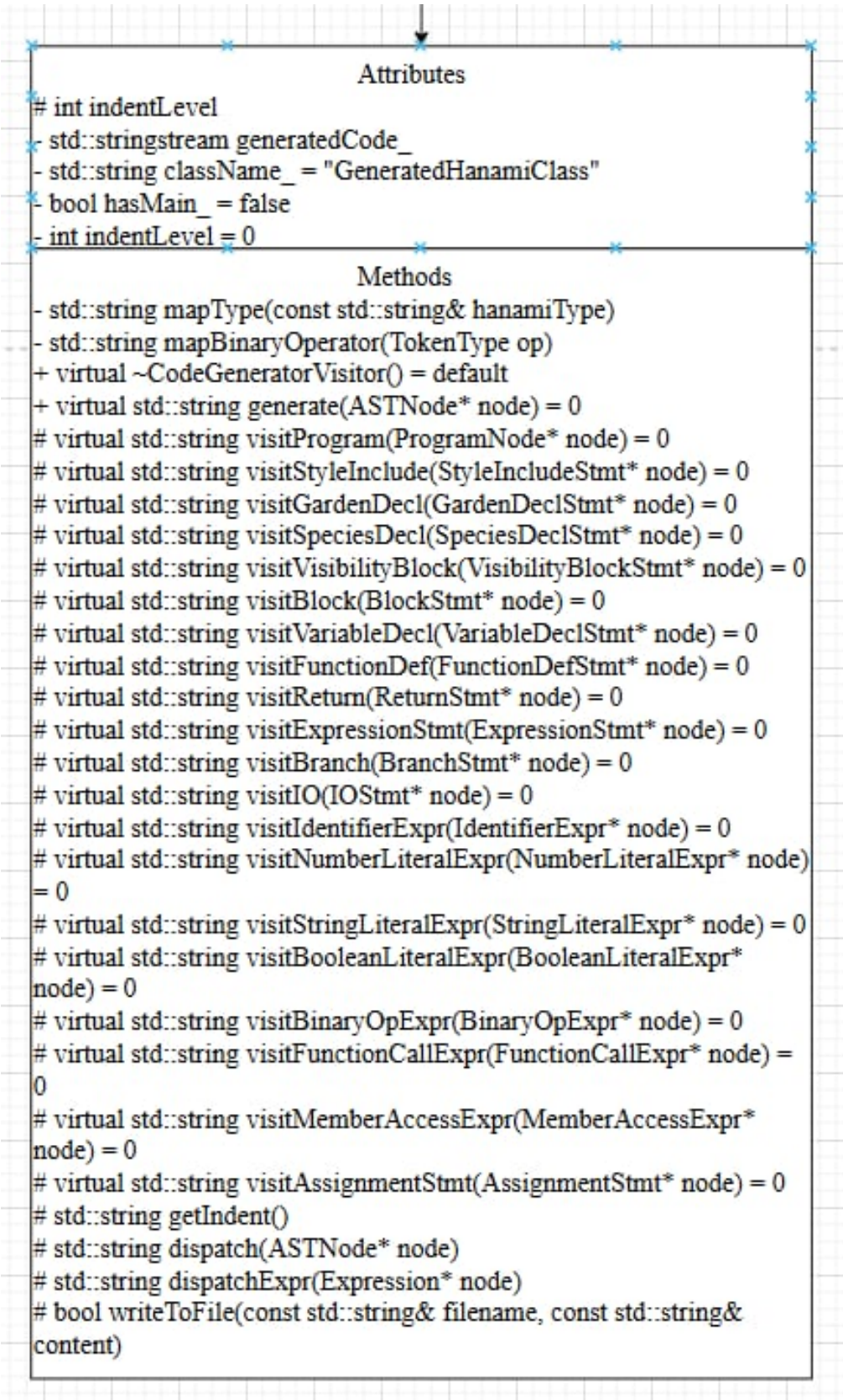



Figure 5: Complete `Lexer` class UML split across six pages to show all attributes, methods, the `Token` struct, and the entire `TokenType` enumeration (from `RIGHT_BRACE` through `SCOPE`).

# G Code Generator Visitor


Attributes
# int indentLevel
- std::stringstream generatedCode_
- std::string className_ = "GeneratedHanamiClass"
- bool hasMain_ = false
- int indentLevel = 0
Methods
- std::string mapType(const std::string& hanamiType)
- std::string mapBinaryOperator(TokenType op)
+ virtual ~CodeGeneratorVisitor() = default
+ virtual std::string generate(ASTNode* node) = 0
# virtual std::string visitProgram(ProgramNode* node) = 0
# virtual std::string visitStyleInclude(StyleIncludeStmt* node) = 0
# virtual std::string visitGardenDecl(GardenDeclStmt* node) = 0
# virtual std::string visitSpeciesDecl(SpeciesDeclStmt* node) = 0
# virtual std::string visitVisibilityBlock(VisibilityBlockStmt* node) = 0
# virtual std::string visitBlock(BlockStmt* node) = 0
# virtual std::string visitVariableDecl(VariableDeclStmt* node) = 0
# virtual std::string visitFunctionDef(FunctionDefStmt* node) = 0
# virtual std::string visitReturn(ReturnStmt* node) = 0
# virtual std::string visitExpressionStmt(ExpressionStmt* node) = 0
# virtual std::string visitBranch(BranchStmt* node) = 0
# virtual std::string visitIO(IOStmt* node) = 0
# virtual std::string visitIdentifierExpr(IdentifierExpr* node) = 0
# virtual std::string visitNumberLiteralExpr(NumberLiteralExpr* node) = 0
# virtual std::string visitStringLiteralExpr(StringLiteralExpr* node) = 0
# virtual std::string visitBooleanLiteralExpr(BooleanLiteralExpr* node) = 0
# virtual std::string visitBinaryOpExpr(BinaryOpExpr* node) = 0
# virtual std::string visitFunctionCallExpr(FunctionCallExpr* node) = 0
# virtual std::string visitMemberAccessExpr(MemberAccessExpr* node) = 0
# virtual std::string visitAssignmentStmt(AssignmentStmt* node) = 0
# std::string getIndent()
# std::string dispatch(ASTNode* node)
# std::string dispatchExpr(Expression* node)
# bool writeToFile(const std::string& filename, const std::string& content)


Attributes
# int indentLevel
- std::stringstream generatedCode_
- std::string currentSpeciesName_
Methods
- std::string mapType(const std::string& hanamiType)
- std::string mapBinaryOperator(TokenType op)
+ virtual ~CodeGeneratorVisitor() = default
+ virtual std::string generate(ASTNode* node) = 0
# virtual std::string visitProgram(ProgramNode* node) = 0
# virtual std::string visitStyleInclude(StyleIncludeStmt* node) = 0
# virtual std::string visitGardenDecl(GardenDeclStmt* node) = 0
# virtual std::string visitSpeciesDecl(SpeciesDeclStmt* node) = 0
# virtual std::string visitVisibilityBlock(VisibilityBlockStmt* node) = 0
# virtual std::string visitBlock(BlockStmt* node) = 0
# virtual std::string visitVariableDecl(VariableDeclStmt* node) = 0
# virtual std::string visitFunctionDef(FunctionDefStmt* node) = 0
# virtual std::string visitReturn(ReturnStmt* node) = 0
# virtual std::string visitExpressionStmt(ExpressionStmt* node) = 0
# virtual std::string visitBranch(BranchStmt* node) = 0
# virtual std::string visitIO(IOStmt* node) = 0
# virtual std::string visitIdentifierExpr(IdentifierExpr* node) = 0
# virtual std::string visitNumberLiteralExpr(NumberLiteralExpr* node) = 0
# virtual std::string visitStringLiteralExpr(StringLiteralExpr* node) = 0
# virtual std::string visitBooleanLiteralExpr(BooleanLiteralExpr* node) = 0
# virtual std::string visitBinaryOpExpr(BinaryOpExpr* node) = 0
# virtual std::string visitFunctionCallExpr(FunctionCallExpr* node) = 0
# virtual std::string visitMemberAccessExpr(MemberAccessExpr* node) = 0
# virtual std::string visitAssignmentStmt(AssignmentStmt* node) = 0
# std::string getIndent()
# std::string dispatch(ASTNode* node)
# std::string dispatchExpr(Expression* node)
# bool writeToFile(const std::string& filename, const std::string& content)


Attributes
# int indentLevel
-- std::stringstream generatedCode_
- std::set<std::string> includes_
- bool hasMain_ = false
Methods
- std::string mapType(const std::string& hanamiType)
- std::string mapBinaryOperator(TokenType op)
+ virtual ~CodeGeneratorVisitor() = default
+ virtual std::string generate(ASTNode* node) = 0
# virtual std::string visitProgram(ProgramNode* node) = 0
# virtual std::string visitStyleInclude(StyleIncludeStmt* node) = 0
# virtual std::string visitGardenDecl(GardenDeclStmt* node) = 0
# virtual std::string visitSpeciesDecl(SpeciesDeclStmt* node) = 0
# virtual std::string visitVisibilityBlock(VisibilityBlockStmt* node) = 0
# virtual std::string visitBlock(BlockStmt* node) = 0
# virtual std::string visitVariableDecl(VariableDeclStmt* node) = 0
# virtual std::string visitFunctionDef(FunctionDefStmt* node) = 0
# virtual std::string visitReturn(ReturnStmt* node) = 0
# virtual std::string visitExpressionStmt(ExpressionStmt* node) = 0
# virtual std::string visitBranch(BranchStmt* node) = 0
# virtual std::string visitIO(IOStmt* node) = 0
# virtual std::string visitIdentifierExpr(IdentifierExpr* node) = 0
# virtual std::string visitNumberLiteralExpr(NumberLiteralExpr* node) = 0
# virtual std::string visitStringLiteralExpr(StringLiteralExpr* node) = 0
# virtual std::string visitBooleanLiteralExpr(BooleanLiteralExpr* node) = 0
# virtual std::string visitBinaryOpExpr(BinaryOpExpr* node) = 0
# virtual std::string visitFunctionCallExpr(FunctionCallExpr* node) = 0
# virtual std::string visitMemberAccessExpr(MemberAccessExpr* node) = 0
# virtual std::string visitAssignmentStmt(AssignmentStmt* node) = 0
# std::string getIndent()
# std::string dispatch(ASTNode* node)
# std::string dispatchExpr(Expression* node)
# bool writeToFile(const std::string& filename, const std::string& content)


Figure 6: Abstract `CodeGeneratorVisitor`.

## H IR-Based Code Generator

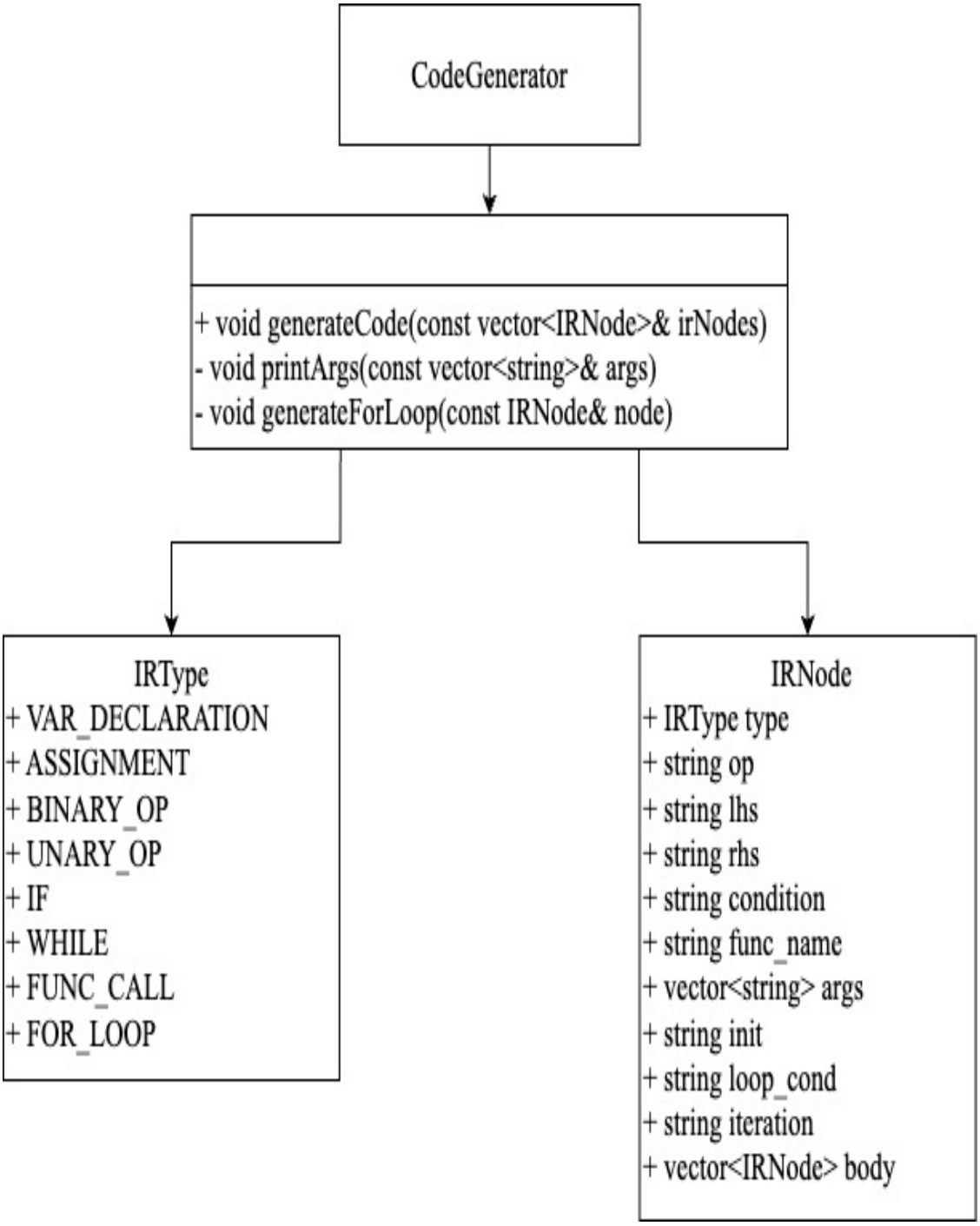



Figure 7: CodeGenerator class overview with IR types and generation routines.

# I Sequence Diagram

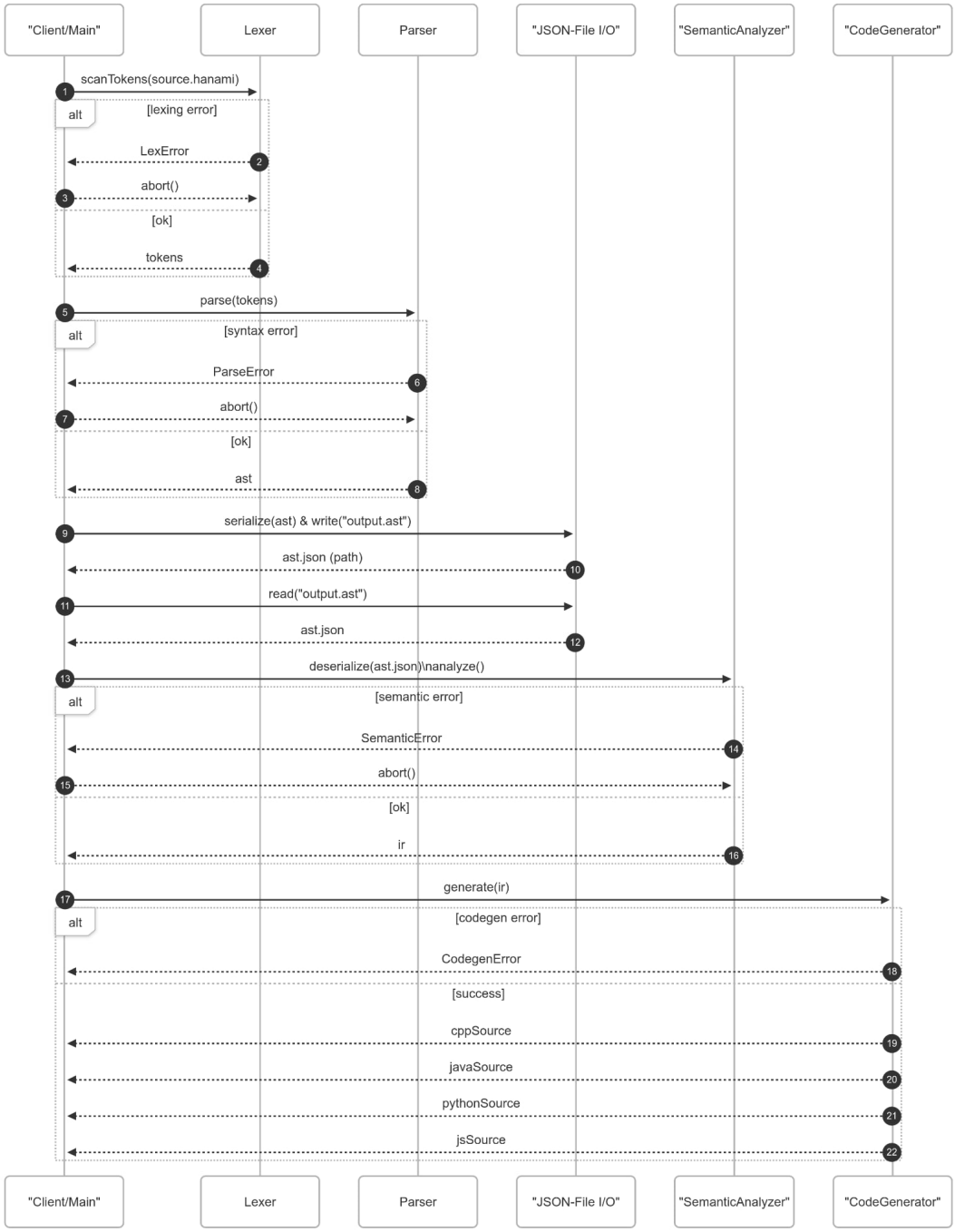



Figure 8: End-to-end sequence diagram depicting interactions between Client/Main, Lexer, Parser, JSON I/O, Semantic Analyzer, and Code Generator.