System Architecture of the Hanami Compiler

April 24, 2025

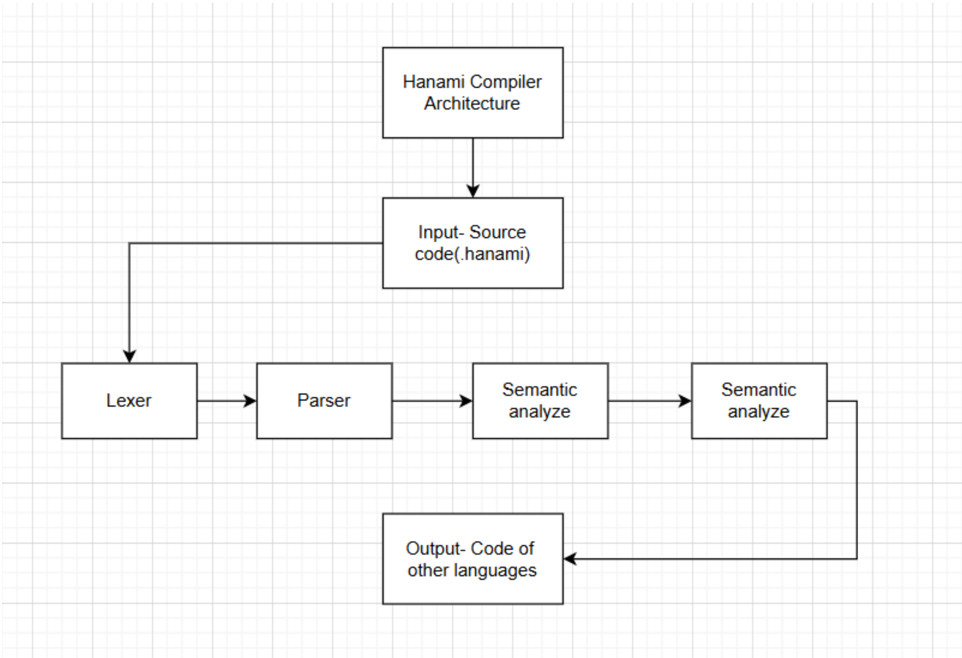

Figure 1: Table of Contents

1 High-Level Architecture Overview

The Hanami compiler transforms Hanami source code (.hanami) into other languages like C++, Java, or Python. The compilation process involves distinct phases, each performing specific transformations. The system is structured into primary components such as the Lexer, Parser, Semantic Analyzer, and Code Generator.

1.1 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 can be executed concurrently for improved performance

1.2 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, or Python Code

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

2 Lexer

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

2.1 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

2.1.1 Lexer Algorithm

The lexer employs a deterministic finite automaton (DFA) approach:

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

3 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.

3.1 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

3.1.1 Parsing Algorithm

The parser uses a recursive descent approach with predictive parsing:

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

3.1.2 AST Structure

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

4 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.

4.1 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

4.1.1 Symbol Table Structure

The symbol table uses a hierarchical structure to represent nested scopes, allowing for efficient symbol lookup and scope management.

4.1.2 Type System

The type system defines:

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

4.1.3 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

5 Code Generator

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

5.1 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

5.1.1 Code Generation Process

The code generation process involves:

1. Traverse the AST
2. Generate code for each node, considering the target language
3. Handle language-specific features
4. Output the generated code

5.1.2 Target Languages

The code generator supports multiple target languages:

- C++
- Java
- Python

6 Support Systems

6.1 Build System

The build system orchestrates the compilation process, managing dependencies and build configurations.

6.1.1 Build System Components

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

6.1.2 Build Targets

The build system provides several targets:

- **clean**: Remove build artifacts
- **build**: Compile the compiler
- **test**: Run test suite
- **install**: Install compiler binaries
- **package**: Create distributable packages

6.1.3 Build Configuration

The build system supports multiple configurations:

- Debug build (with debugging symbols)
- Release build (optimized)
- Cross-compilation settings
- Feature toggles

6.2 Common Utilities

The `common/` directory contains shared utilities used across compiler components.

6.2.1 Utility Components

- **Error Handling**: Consistent error reporting infrastructure
- **Source Location Management**: Tracking file, line, and column information
- **Memory Management**: Custom allocators for compiler data structures
- **Diagnostics**: Error and warning message formatting

6.2.2 Data Structures

The common utilities provide specialized data structures:

- **Symbol Tables**: Efficient lookup structures
- **Abstract Syntax Trees**: Node hierarchies
- **String Interning**: Memory-efficient string storage

6.2.3 I/O Facilities

The utilities include I/O facilities for:

- Source file reading
- Output file generation
- Error stream management
- Console interaction

7 Inter-Component Communication

7.1 Data Exchange Formats

Components communicate through well-defined data structures:

- **Token Stream**: Between lexer and parser
- **Abstract Syntax Tree**: Between parser and semantic analyzer
- **Annotated AST**: Between semantic analyzer and code generator
- **C++, Java, Python Code**: Output from code generator

7.2 Error Handling

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

7.3 Progress Tracking

The compiler tracks compilation progress for reporting:

- **Phase Completion**: Indication of completed compilation phases
- **Statistics**: Time spent in each phase, memory usage
- **Progress Indicators**: For long-running operations

8 Compilation Pipeline

The Hanami compilation process follows a sequence of distinct phases:

8.1 Source Processing Phase

1. **Source Reading**: Loading source files into memory
2. **Lexical Analysis**: Conversion of source code to token stream

8.2 Syntax Processing Phase

1. **Parsing**: Construction of abstract syntax tree
2. **AST Verification**: Validation of AST structural integrity
3. **AST Transformation**: Normalization of AST structures

8.3 Semantic Processing Phase

1. **Symbol Collection**: Building symbol tables
2. **Type Analysis**: Type checking and inference
3. **Semantic Validation**: Enforcing language semantic rules

8.4 Code Generation Phase

1. **Code Generation**: Converting AST to target language code
2. **Code Formatting**: Formatting the generated code

9 System Architecture Evaluation

9.1 Performance Considerations

The architecture is designed with performance in mind:

- **Incremental Processing**: Enabling parallel compilation when possible
- **Memory Efficiency**: Careful management of large data structures
- **Processing Time**: Optimization of time-intensive operations

9.2 Scalability Aspects

The architecture supports language and compiler evolution:

- **Feature Extensibility**: Clear extension points for new language features
- **Target Platform Addition**: Modular approach to supporting new targets
- **Optimization Expansion**: Framework for adding new optimization passes

9.3 Maintainability Factors

The architecture promotes maintainability:

- **Component Isolation**: Minimizing inter-component dependencies
- **Interface Stability**: Clear contracts between components
- **Testing Support**: Design that facilitates comprehensive testing

9.4 Quality Assurance

The architecture includes provisions for quality assurance:

- **Verification Points**: Strategic checks throughout the compilation pipeline
- **Diagnostic Capabilities**: Extensive error detection and reporting
- **Tracing Infrastructure**: For debugging and performance analysis