Mini Java Compiler Report

Weihao Ni, Yulong Ma

March 14, 2025

Abstract

This report presents the design and implementation of a Mini Java Compiler that translates Mini Java code into x86-64 assembly. The implementation follows a structured approach with three key phases: parsing, type checking, and code generation . The Visitor pattern is used throughout the compiler to traverse Abstract Syntax Trees (ASTs) and apply transformations in a modular fashion. This document provides a briefly explanation of the compiler's architecture, implementation, debugging process, and testing results.

1 Introduction

Mini Java is a simplified project aims to implement a compiler that translates Mini Java programs into executable x86-64 assembly code . The compiler consists of three major phases:

- Syntax Parsing – Constructs an Abstract Syntax Tree (AST) from Mini Java source code.
- Type Checking – Ensures semantic correctness by verifying type constraints and method compatibility.
- Code Generation – Produces assembly code with proper function calls, memory management, and stack organization.

Each phase is implemented using the Visitor pattern , which enables separation of concerns and modular transformation of AST nodes .

2 Project Structure

The project is implemented in three main Java files:

2.1 Syntax Parsing (`Syntax.java`)

The Syntax.java file defines the Abstract Syntax Tree (AST) representation for Mini Java programs. Each AST node corresponds to a language construct , such as expressions, statements, method calls, and object instantiations. The file also defines the TVisitor interface , allowing different phases to traverse and manipulate the AST.

2.2 Type Checking (`TransferPT.java`)

The TransferPT.java file performs semantic analysis and type checking. It ensures:

- Variable and method declarations are valid.
- Type correctness in assignments, expressions, and method calls.
- Inheritance rules are followed, including method overriding and attribute access.

It maintains a symbol table to track variable declarations, types, and scopes, ensuring correct type usage throughout the program.

2.3 Code Generation (`Compile.java`)

The Compile.java file translates the typed AST into x86-64 assembly code . It handles:

- Memory allocation for objects and variables.
- Stack frame management for function calls and local variables.
- Virtual method tables (VMTs) for dynamic method dispatch.

The generated assembly code follows System V calling conventions , ensuring correct function execution.

3 Implementation Details

3.1 Visitor Pattern in the Compiler

The Visitor pattern is used throughout the compiler to process AST nodes in different phases:

- Parsing – Constructs the AST from source code.
- Type Checking – Uses a visitor to annotate nodes with type information.
- Code Generation – Uses a visitor to generate corresponding assembly instructions.

This approach allows each phase to operate independently, ensuring modularity and maintainability .

3.2 Syntax Parsing and AST Representation

The syntax parsing phase translates Mini Java source code into an Abstract Syntax Tree (AST), which serves as the core data structure for subsequent type checking and code generation. This phase ensures that source code follows correct grammatical rules and converts it into a structured tree format.

- Abstract Syntax Tree (AST) Structure

The AST is implemented in ‘Syntax.java‘, where each node corresponds to a specific language construct, such as expressions, statements, or declarations.

Key design choices include: Node Hierarchy: Each AST node extends from a common base class (‘TExpr‘ for expressions, ‘TStmt‘ for statements), ensuring uniform traversal; Strong Typing: The AST differentiates between constants (‘TEcst‘), variables (‘TEvar‘), method calls (‘TEcall‘), object creation (‘TEnew‘), and control flow statements;

Visitor Pattern Integration: Each AST node implements an ‘accept‘ method, allowing type checking and code generation to process nodes without modifying their structure. Using the Visitor pattern here ensures modularity, allowing different phases of the compiler to process AST nodes independently while maintaining clear separation of concerns.

- Expression and Statement Representation

Expressions (‘TExpr‘) represent computations and include: Arithmetic operations (‘TEbinop‘) such as addition and multiplication; Boolean operations (‘TEcomp‘, ‘TEneg‘) used for comparisons and logical negation; Method calls (‘TEcall‘) representing function invocations with arguments.

Statements (‘TStmt‘) define program execution flow and include: Variable assignments (‘TSassign‘) storing computed values; Conditionals (‘TSif‘) and loops (‘TSwhile‘) handling control flow; Method return statements (‘TSreturn‘) ensuring correct function outputs.

By structuring the AST this way, Mini Java programs can be easily analyzed, transformed, and converted into executable machine code.

3.3 Type Checking and Semantic Analysis

The second phase ensures semantic correctness by verifying types, resolving variable references, and enforcing Mini Java rules. This is implemented in ‘TransferPT.java‘, which transforms the AST into a typed intermediate representation.

- Symbol Tables and Scope Management

A symbol table is used to track declared variables, methods, and classes. It is implemented using hash maps: Class Table (‘classMap‘) stores defined classes, their attributes, and methods.

During traversal, the visitor ensures that every identifier has been declared before use, preventing issues such as: Undefined variables or methods (e.g., calling an undeclared function); Incorrect argument types in method calls; Inheritance conflicts, such as illegal method overrides.

- Handling Inheritance and Overloading

Inheritance is validated by checking that: Subclasses correctly extend existing classes (‘extends‘ verification); Overridden methods match their parent signatures (same return type and arguments); Parent class attributes are correctly inherited and referenced in subclass methods.

- Type Propagation and Checking

Each expression node (‘TExpr‘) is annotated with a computed type based on: Literal types (integers, booleans); Variable lookups (fetching stored type information); Arithmetic and logical operations, enforcing valid operand types; Method call resolution, checking return types against expected values.

Using type inference and validation, this phase eliminates semantic errors before code generation, ensuring only well-formed programs reach the final compilation step.

3.4 Stack Management and Memory Allocation

The stack frame layout is managed efficiently in the computeLocalVarOffsets function. It assigns stack offsets to:

- Method parameters stored at +24(%rbp), +32(%rbp), ... (accounting for ‘this‘ pointer).
- Local variables stored at -8(%rbp), -16(%rbp), ...

Stack space is allocated dynamically based on the number of local variables, ensuring correct stack alignment before function calls.

3.5 Object Representation and Method Dispatch

Each Mini Java class has a descriptor table , which contains:

- A pointer to its parent class descriptor (for inheritance).
- A method table (VMT) storing function pointers for dynamic dispatch.

3.6 Generating Function Calls

- Static function calls use direct jumps (‘call label‘).
- Dynamic method calls retrieve function pointers from the method table

4 Challenges and Debugging

4.1 Incorrect Stack Offsets

One major challenge was incorrect offset calculations , causing invalid memory accesses. This was fixed by:

- Ensuring ‘computeLocalVarOffsets‘ assigns valid offsets before ‘visit(TEvar)‘.
- Before allocating memory using mymalloc, you need to recursively find the largest attribute offset to allocate memory to avoid hard-to-track segmentation faults.
- We need to ensure all objects are properly allocated before accessing fields or calling methods.

4.2 Handling Special Classes

In mini-java, we need to provide special external support for ’String’ and ’System’ classes.

- For ’String’ class, we added ’equals’ method to it. For the String constants , our initial approach was to add a class descriptor to each String so that they can be compared and call methods, but we realized that this would affect the concatenation of numbers and strings. In order to minimize the impact of String, we perform special checks on the String class in compile to specifically call ’cast’ and ’equals’.
- For ’System’, we added ’print’ methods and added special checks, classifying calling ’print’ as ’TEprint’ instead of ’TEcall’ to facilitate the processing logic during compilation. And during the compilation, different parameters of print are processed differently to handle different inputs.

4.3 Challenges with Inheritance and Method Dispatch

Handling subclass inheritance and method resolution presented significant challenges in ensuring correct memory layout, function calls, and attribute access. Since Mini Java allows subclasses to override parent methods, it was crucial to correctly update the Virtual Method Table (VMT) to support dynamic dispatch.

- The VMT serves as a lookup table for method calls, ensuring that overridden methods in a subclass correctly replace parent method entries. Additionally, newly declared methods had to be inserted at the correct offsets while maintaining compatibility with inherited ones. Ensuring that objects referenced their correct class descriptors was essential for proper method resolution at runtime.
- Another challenge was managing attribute offsets in an inheritance hierarchy. Each subclass must inherit its parent’s attributes while also introducing its own. This required careful tracking of memory offsets to prevent incorrect attribute lookups or overwriting parent attributes. Handling multiple levels of inheritance required ensuring that even deeply inherited attributes were assigned consistent offsets.
- Initially, method lookups in subclasses failed due to incorrect VMT resolution. This was resolved by ensuring that subclasses properly inherited their parent’s VMT pointer while implementing runtime method resolution using function pointer dereferencing. By structuring the VMT correctly and ensuring each class descriptor pointed to the appropriate function entries, we successfully enabled dynamic method dispatch without breaking subclass behavior.

- Object instantiation (‘TEnew‘) also required careful memory allocation and class descriptor setup. Incorrectly assigned descriptors led to segmentation faults during method calls or field access. To resolve this, we ensured that allocated memory correctly accounted for all attributes—including those inherited—while setting up the proper class descriptor pointers. This allowed objects to support both correct attribute access and polymorphic behavior in method calls.

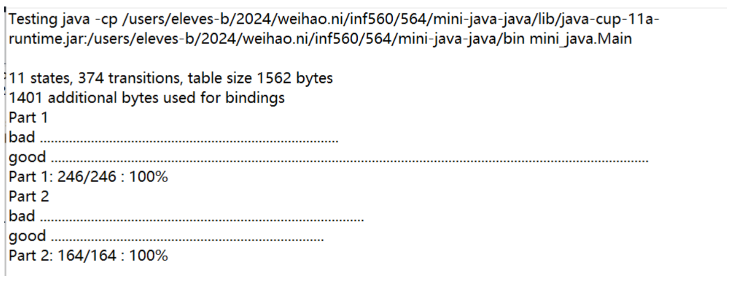
5 Testing and Results

The compiler was validated with a set of Mini Java programs. Testing ensured:

- Correct parsing and AST construction.
- Type checking errors were detected properly.
- Generated assembly code produced correct results.

5.1 Execution Tests

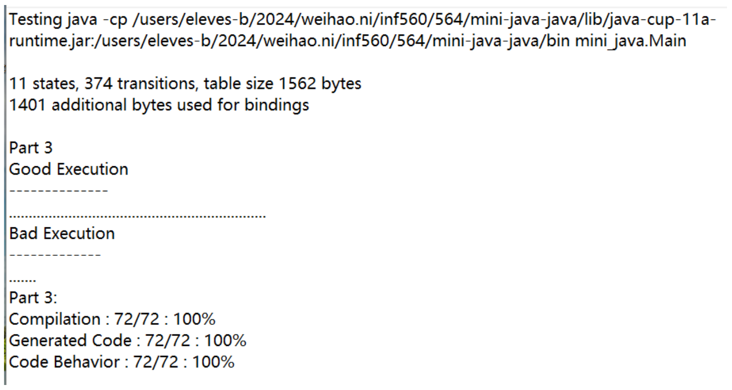
Below are the test results showing successful compilation and execution:



```
Testing java -cp /users/eleves-b/2024/weihao.ni/inf560/564/mini-java-java/lib/java-cup-11a-
runtime.jar:/users/eleves-b/2024/weihao.ni/inf560/564/mini-java-java/bin mini_java.Main

11 states, 374 transitions, table size 1562 bytes
1401 additional bytes used for bindings
Part 1
bad ...................................................................................
good ..................................................................................................................................................................
Part 1: 246/246 : 100%
Part 2
bad ..........................................................................................
good ............................................................................
Part 2: 164/164 : 100%
```

Figure 1: Successful Compilation and Execution of Part1&2 Test Cases



```
Testing java -cp /users/eleves-b/2024/weihao.ni/inf560/564/mini-java-java/lib/java-cup-11a-
runtime.jar:/users/eleves-b/2024/weihao.ni/inf560/564/mini-java-java/bin mini_java.Main

11 states, 374 transitions, table size 1562 bytes
1401 additional bytes used for bindings

Part 3
Good Execution
--------------
........................................................................
Bad Execution
-------------
.......
Part 3:
Compilation : 72/72 : 100%
Generated Code : 72/72 : 100%
Code Behavior : 72/72 : 100%
```

Figure 2: Successful Compilation and Execution of Part3 Test Cases

5.2 Debugging in GDB

We used GDB (GNU Debugger) to inspect register values and memory states for debugging:

6 Conclusion

This project successfully implements a Mini Java compiler that translates Mini Java programs into executable x86-64 assembly code . Despite challenges related to stack offsets, method dispatch, memory management, and String processing, the final implementation is capable of correctly compiling and executing Mini Java programs.