

Compiler Project Report Submission

ORCHESTRA: A Modular Multi-Backend Compiler for Flexible Execution and Code Generation

Subject: Compiler Project Report Submission
Course Title: Compiler Design Laboratory
Course Code: CSE 3212
Submission Date: April 6, 2026

Department of Computer Science and Engineering

Khulna University of Engineering and Technology

Abstract

ORCHESTRA is a custom programming language built to demonstrate a complete compiler pipeline in a small and practical form. The project uses Flex for lexical analysis and Bison for parsing, then builds a shared abstract syntax tree that supports three outputs: direct execution through an AST interpreter, execution through a stack-based virtual machine, and an equivalent C++ view for debugging and presentation.

The project highlights the full flow from source code to execution while also showing backend equivalence, clean language design, and extensibility. Although compact, ORCHESTRA supports functions, variables, control flow, formatted output, collections, object-oriented constructs, and a lightweight pointer model. The result is a compiler project that is easy to explain, easy to demo, and broad enough to show the full potential of the system.

1 Introduction

ORCHESTRA was developed to show how a programming language can be designed, parsed, and executed through multiple backends within one project. The aim was not only to create a custom language, but also to build a system that clearly demonstrates how compiler components work together in practice.

The main goals of the project are straightforward: design a readable language, implement a complete frontend using Flex and Bison, build a shared AST, and run the same program through three different execution paths. These three paths are the AST interpreter, the VM backend, and the C++ emitter used for presentation and debugging.

The scope of the project is summarized by the following pipeline:

Source Code \rightarrow Tokens \rightarrow AST \rightarrow AST Interpreter / VM Backend / C++
Emitter

This report is kept intentionally short. It focuses on what the project does, how it is structured, and which language features make it useful as a compiler project.

2 Language Design — ORCHESTRA

ORCHESTRA is presented here in the same formal style as the initial proposal: short categories, compact tables, and direct examples. The final language keeps the original identity of the proposal, but now includes the full implemented feature set.

Keywords and Their C Equivalents

ORCHESTRA Keyword	Meaning	C Equivalent	Example
flow	Function definition	function	flow add {}
take	Function parameters	parameters	take x, y
emit	Output with newline	printf / return value output	emit sum
note	Variable declaration	variable	note x = 5
fixed	Constant declaration	const	fixed PI = 3.14
stage	Assignment step	assignment	stage s = x + y
ensemble	Structured data	struct	ensemble Point {}
play	Output without forced newline	printf	play x
branch	Conditional statement	if	branch (x == 0)
elsewise	Else block	else	elsewise {}
repeat	Loop	while	repeat (i < n)
score	For-loop construct	for	score (...)
symphony	Class-like declaration	class	symphony Child extends Parent {}
this	Current object context	this	this.value
super	Parent access	base / super	super.init()
stagethru	Pointer write-through	*p = value	stagethru p = 10
return	Return value from a flow	return	return x + 1
break	Exit current loop	break	break;
continue	Skip to next iteration	continue	continue;
extends	Inheritance declaration	extends	symphony C extends P

Data Types Supported

ORCHESTRA Type	Description	Example
<code>int</code>	Integer numbers	10
<code>float</code>	Decimal numbers	3.14
<code>bool</code>	Boolean values	<code>true</code> , <code>false</code>
<code>string</code>	Text data	"hello"
<code>array</code>	Indexed collection	[1, 2, 3]
<code>map</code>	Key-value collection	<code>map()</code>
<code>set</code>	Membership collection	<code>set()</code>
<code>pointer</code>	Reference to a named variable	<code>&x</code>
<code>instance</code>	Object of an ensemble or symphony type	<code>Point(1, 2)</code>

Strings support escape sequences such as `\n`, `\t`, `\\`, and `\"`. Numeric literals support both standard decimal form and scientific notation such as `1e9` and `2.5e-3`.

Operators

Operator	Purpose	C Equivalent	Example
<code>+</code>	Addition	<code>+</code>	<code>a + b</code>
<code>-</code>	Subtraction	<code>-</code>	<code>a - b</code>
<code>*</code>	Multiplication	<code>*</code>	<code>a * b</code>
<code>/</code>	Division	<code>/</code>	<code>a / b</code>
<code>==</code>	Equality	<code>==</code>	<code>x == y</code>
<code>!=</code>	Inequality	<code>!=</code>	<code>x != y</code>
<code><</code> , <code><=</code>	Less-than comparison	<code><</code> , <code><=</code>	<code>x < y</code>
<code>></code> , <code>>=</code>	Greater-than comparison	<code>></code> , <code>>=</code>	<code>x > y</code>
<code>&&</code>	Logical AND	<code>&&</code>	<code>a && b</code>
<code> </code>	Logical OR	<code> </code>	<code>a b</code>
<code>!</code>	Logical NOT	<code>!</code>	<code>!flag</code>
<code>&x</code>	Address-of	reference / address-of	<code>&x</code>
<code>deref(p)</code>	Dereference	<code>*p</code>	<code>deref(p)</code>

The language also supports **comparison chaining**, allowing expressions such as `1 < 2 < 3` and `a == b != c` to be evaluated as true logical chains.

Operator precedence from highest to lowest: unary `!` and `-`, then `*` and `/`, then `+` and `-`, then comparisons (`<` `<=` `>` `>=` `==` `!=`), then `&&`, then `||`.

Control Structures

Conditional Statement

Feature	Description
<code>branch</code>	Conditional execution
<code>elsewise</code>	Alternative execution path

Looping Structure

Feature	Description
<code>repeat</code>	Iterative loop structure
<code>score</code>	For-style loop with init, condition, and step
<code>break</code>	Exit current loop
<code>continue</code>	Move to next iteration

The `score` loop supports optional initialization and step expressions; any or all three parts may be omitted. If the condition is omitted, the loop runs until terminated with `break`. A `continue` statement transfers control to the step expression before the next iteration.

Functions (Flows)

Functions in ORCHESTRA are called **flows**. A flow consists of:

- input parameters
- a sequence of computation stages
- an optional return value

```
flow add take(a, b) {  
    return a + b;  
}
```

At program level, the language supports top-level flows and uses `main` as the normal entry point when present. If no `main` flow is defined, execution begins at the first top-level flow in the program. The `take(...)` parameter part may also be omitted for flows with no parameters.

Variables, Output, and Assignment

Construct	Purpose
<code>note x = expr;</code>	Mutable variable declaration
<code>fixed x = expr;</code>	Immutable constant declaration
<code>stage x = expr;</code>	Reassignment of existing variable
<code>emit expr;</code>	Output followed by newline
<code>play expr;</code>	Output without automatic newline
<code>play fmt, args...;</code>	Formatted output
<code>return expr;</code>	Return from a flow
<code>return;</code>	Empty return

Formatted output through `play` supports `%d`, `%f`, `%s`, and `%b`. Constants declared with `fixed` cannot be reassigned later and support only scalar types (`int`, `float`, `bool`, `string`).

Collection Features

Feature	Description
<code>[a, b, c]</code>	Array literal
<code>target[index]</code>	Indexing for arrays, maps, and sets
<code>stage arr[i] = x;</code>	Indexed assignment
<code>array(n)</code>	Create array of size <code>n</code>
<code>push, pop, resize</code>	Built-in array operations
<code>map(), get, put</code>	Map construction and update helpers
<code>set(), add, has, del, keys</code>	Set and shared collection helpers

Indexing behavior depends on the target collection: arrays return the element at a position, maps return the value for a key, and sets return a boolean membership result. Map and set keys follow a string-key model.

Object-Oriented Features

Feature	Description
<code>ensemble</code>	Struct-like type with fields
<code>symphony</code>	Class-like type with fields and methods
<code>extends</code>	Inheritance support
<code>this</code>	Current instance reference
<code>super</code>	Parent constructor or method access
<code>obj.field</code> / <code>obj.method()</code>	Field access and instance method calls
<code>TypeName.method(args)</code>	Static-style method call on a named type
<code>stage obj.field = expr;</code>	Field assignment
<code>TypeName(...)</code>	Constructor-like instance creation

Inheritance is supported through `extends`. A `symphony` may use `super(args)` for constructor chaining and `super.method(args)` for parent method access. If a type defines `init`, it acts as a user-defined constructor.

Pointer Features

Feature	Description
<code>&x</code>	Create a pointer to variable <code>x</code>
<code>deref(p)</code>	Read the value referenced by pointer <code>p</code>
<code>stagethru p = expr;</code>	Update the variable referenced by pointer <code>p</code>

Pointers can be used directly in expressions, for example `deref(p) + 3`. The pointer model works with `int`, `float`, `bool`, and `string` values.

Additional Language Features

- single-line comments using `//` and multiline comments using `/* ... */`
- scientific notation such as `1e9` and `2.5e-3`
- lexical scoping and shadowing through nested blocks
- formatted output using `play`
- recursion through flow calls, with runtime recursion-limit checks
- equivalent execution behavior across AST and VM backends

C++ Emit Mode

The `-emit=cpp` mode generates a readable C++-style representation of the AST without executing the program. It supports two styles: `cpp` (using `auto` and `emit(...);`) and `pseudo` (using `var` and `print(...);`). A keyword legend is included to map ORCHESTRA constructs to their C++ equivalents.

3 System Architecture

The ORCHESTRA compiler follows a shared-frontend, multi-backend design. Source code is first processed by the lexer and parser to produce a single abstract syntax tree. That AST is then reused by three different paths: direct execution through the AST interpreter, compilation into bytecode for the VM backend, and translation into an equivalent C++ representation for visualization.

This architecture keeps the frontend common while allowing multiple execution strategies to coexist in the same project. It also makes feature testing easier because the AST backend serves as the reference model and the VM backend can be validated against it.

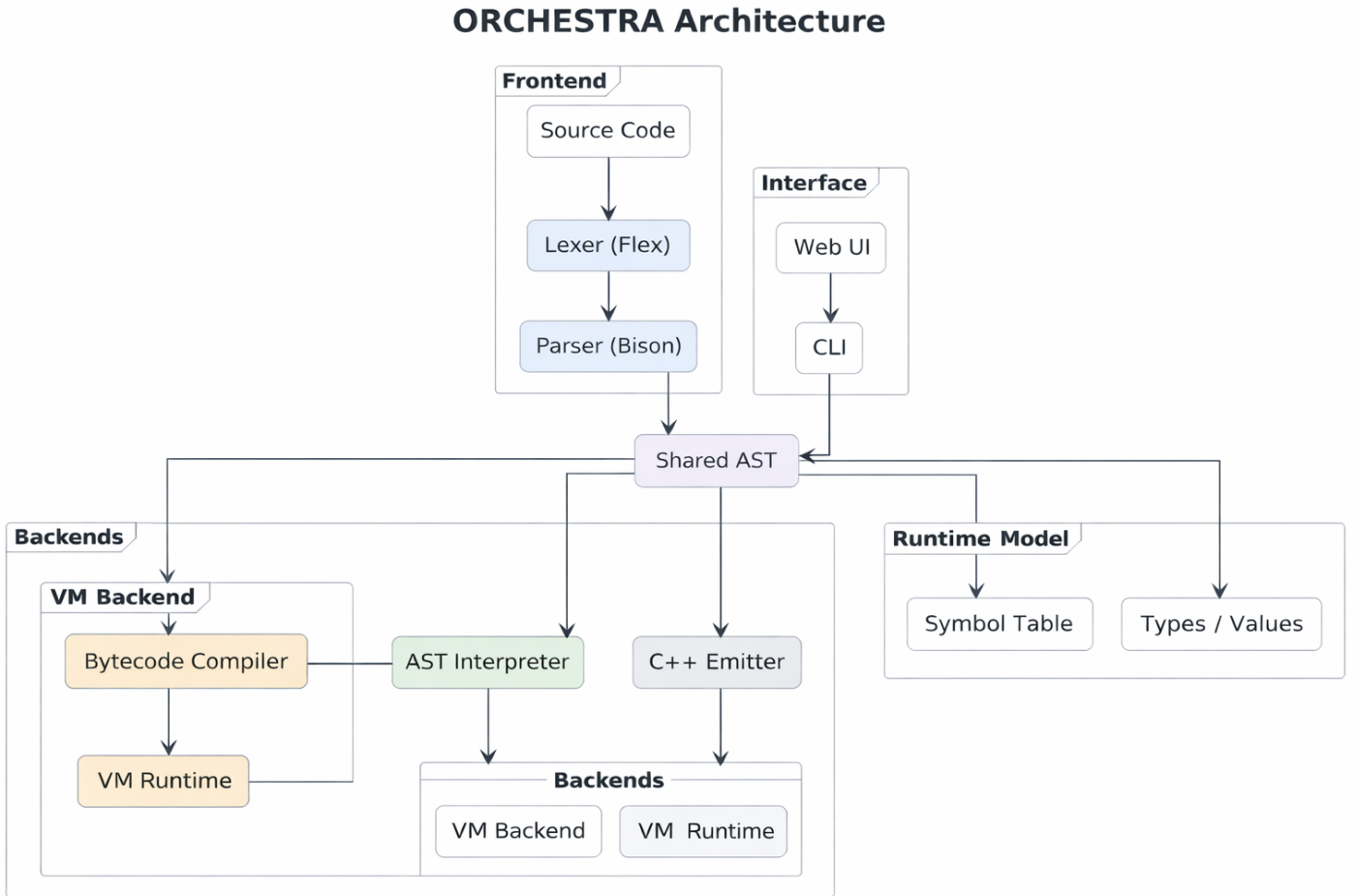


Figure 1: System architecture of the ORCHESTRA compiler.

4 Code Execution Flow

The execution path of a program begins with source input and ends in one of three outputs. The lexer identifies tokens, the parser builds the AST, and the selected backend determines the final execution behavior. In AST mode, the tree is interpreted directly. In VM mode, the AST is converted into bytecode and executed by the virtual machine. In emit mode, the AST is translated into equivalent C++-style code for presentation and debugging.

This flow shows how one frontend can support multiple backend targets without changing the user-facing language.

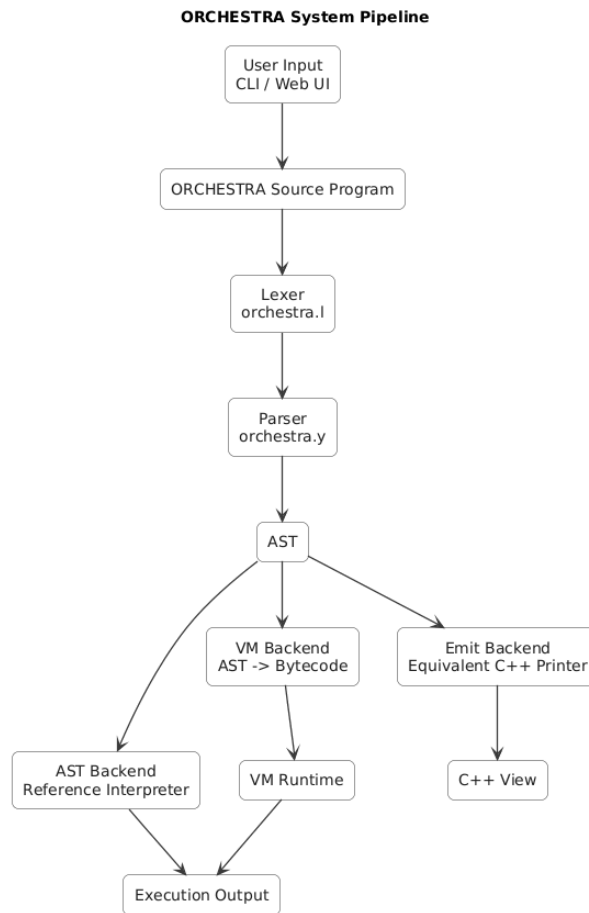


Figure 2: Code execution flow in the ORCHESTRA compiler.

5 Implementation

The implementation is divided across a small set of core files, each with a clearly defined role in the compiler pipeline.

File	Role
<code>orchestra.l</code>	Lexer specification used for token generation.
<code>orchestra.y</code>	Parser grammar used to build the AST and drive compilation modes.
<code>interpreter.h</code>	Shared declarations for AST nodes, values, and runtime structures.
<code>interpreter.c</code>	AST interpreter implementation and core runtime behavior.
<code>symbol_table.h</code> / <code>symbol_table.cpp</code>	Scope and symbol management for variables and declarations.
<code>vm_backend.cpp</code>	AST-to-bytecode compilation for the VM backend.
<code>bytecode.h</code> / <code>bytecode.cpp</code>	Bytecode definitions, VM values, and bytecode execution engine.
<code>cpp_printer.h</code> / <code>cpp_printer.cpp</code>	Equivalent C++ generation for emit mode.
<code>flow_registry.h</code> / <code>flow_registry.cpp</code>	Registration and lookup of defined flows.
<code>webui_server.py</code>	Python server that connects the browser UI to <code>orchestra.exe</code> .
<code>webui/index.html</code> , <code>app.js</code> , <code>styles.css</code>	Browser-based playground interface for running and viewing programs.

6 Error Handling

The compiler includes error checks across lexical analysis, parsing, semantic validation, runtime execution, and VM execution. The goal is to detect errors at the earliest possible stage and report them in a clear form.

Lexical Errors

Error Type	Example	Handling
Invalid character / unknown token	<code>emit x @ y;</code>	Reports an unexpected character with line information.
Unterminated string literal	<code>note msg = "hello;</code>	Reports an unterminated string literal when closing quote is missing.
Invalid number format	<code>emit 12.3.4;</code>	Rejects malformed numeric literals during lexing.

Syntax Errors

Error Type	Example	Handling
Missing semicolon	<code>note a = 5</code>	Parser reports a syntax error near the next unexpected token.
Malformed construct	<code>branch (x > 5 {</code>	Parser reports invalid structure such as missing delimiters or parentheses.

Semantic Errors

Error Type	Example	Handling
Undefined variable	<code>emit x;</code>	Reports use of a variable before declaration.
Redeclaration in same scope	<code>note x = 1; note x = 2;</code>	Reports variable redeclaration in the same scope.
Invalid assignment target	<code>stage (a + b) = 3;</code>	Rejects non-assignable expressions as assignment targets.

Runtime and Function Call Errors

Error Type	Example	Handling
Division by zero	<code>emit a / 0;</code>	Reports runtime division by zero.
Recursion limit exceeded	Deep recursive calls	Stops execution when recursion exceeds the configured limit.
Invalid condition type	<code>branch("hi") {}</code>	Requires the condition to be boolean or follow truthiness rules.
Calling undefined flow	Undefined function call	Reports unknown flow at runtime.
Wrong number of arguments	Incorrect call arity	Reports incorrect argument count.

VM / Backend Errors

Error Type	Handling
Stack underflow / overflow	VM checks stack bounds before push and pop operations.
Invalid jump target	VM validates jump addresses during execution.
Backend mismatch	AST backend is treated as reference and compared against VM output during equivalence testing.

7 Testing & Validation

The project includes both single-backend testing and backend-equivalence testing. The test setup is organized through batch scripts so that the compiler can be validated quickly during development and demonstration.

Test Suite Organization

Script	Purpose
<code>run_tests.bat</code>	Runs the AST-oriented test suite.
<code>run_vm_tests.bat</code>	Runs AST and VM backends and compares their outputs.
<code>run_emit_cpp_tests.bat</code>	Supports manual validation of the C++ emit output.

Equivalence Testing Strategy

The AST interpreter is treated as the reference backend. For VM validation, the same source program is executed in both AST mode and VM mode, and the produced outputs are compared using the Windows `fc` command. This makes it possible to detect backend mismatches quickly and confirm that the VM preserves the intended language semantics.

Test Categories

- expressions and arithmetic evaluation
- control flow and loop behavior
- recursion
- collections: arrays, maps, and sets
- object-oriented features and inheritance
- pointer operations
- lexical scoping and shadowing

Results Summary

Category	Status	Remarks
Expressions	Pass	Arithmetic, logical operations, and chaining validated.
Control Flow	Pass	Branching, repeat, <code>score</code> , <code>break</code> , and <code>continue</code> validated.
Recursion	Pass	Recursive flow execution validated with runtime checks.
Collections	Pass	Arrays, maps, sets, indexing, and helpers validated.
OOP Features	Pass	Fields, methods, inheritance, <code>this</code> , and <code>super</code> validated.
Pointers	Pass	Address-of, dereference, and write-through behavior validated.
Shadowing	Pass	Nested lexical scope behavior validated.

8 Web UI Playground

The project also includes a local Web UI for interactive demonstration. This makes the compiler easier to present because source code can be entered in the browser and executed using the same backend logic as the command-line version.

Execution Model

The Web UI does not implement a separate compiler. Instead, it acts as a wrapper around the existing executable, which ensures consistency between the browser demo and the command-line system.

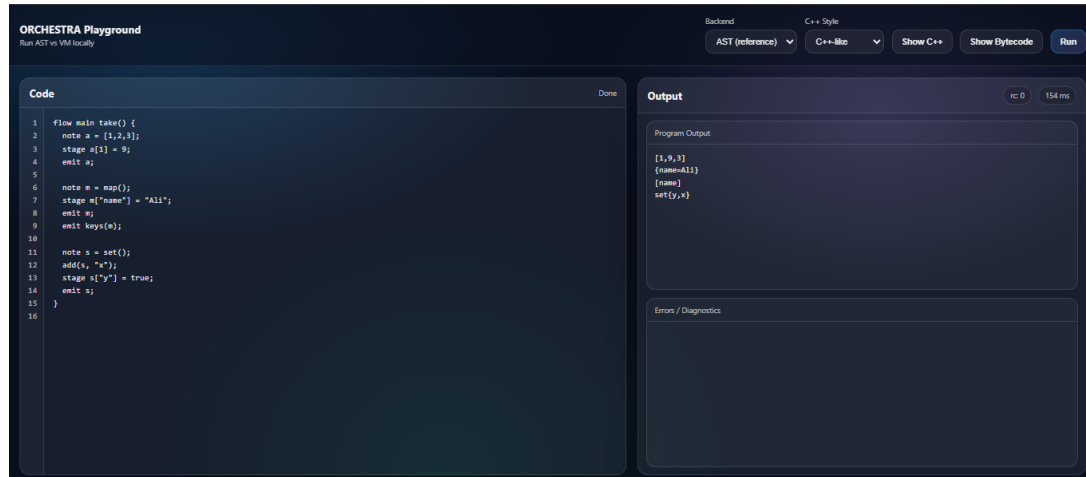


Figure 3: Web UI playground for the ORCHESTRA compiler.

9 Results & Demo

This section presents one representative program and its observed output.

Demonstration Example

Input Program

```
flow main take() {
  note value = 7;
  note ref = &value;
  stagethru ref = 12;

  note nums = [1, 2, 3];
  stage nums[1] = deref(ref);

  branch (nums[1] > 10) {
    emit "updated";
  } else {
    emit "unchanged";
  }

  emit nums;
  emit deref(ref);
}
```

Observed Output

```
updated
[1, 12, 3]
12
```

Validated In

Mode	Status
AST	Pass
VM	Pass

Outcome

The demo confirms identical behavior across AST and VM execution while covering pointer update, array mutation, conditional control flow, and formatted runtime output.

10 Conclusion

The ORCHESTRA project demonstrates a complete compiler workflow in a compact but feature-rich system. It combines a Flex/Bison frontend, a reference AST interpreter, a stack-based virtual machine, and a C++ emit mode within a single architecture. The language itself supports core programming constructs together with collections, object-oriented features, pointers, scientific notation, multiline comments, comparison chaining, recursion, and scoped execution.

The project is strengthened by backend equivalence testing, structured error handling, and a browser-based playground for demonstration. As a result, ORCHESTRA is not only a working custom language, but also a clear and practical example of compiler design and implementation.