Bleach: A Programming Language for Teaching Compilers

Victor Costa
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil
vmmc2@cin.ufpe.br

Leopoldo Teixeira
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil
lmt@cin.ufpe.br

Abstract

Programming languages designed as teaching tools for undergraduate Compilers courses occupy a constrained design space. Existing options such as COOL and MiniJava are deliberately minimal and statically typed, while ChocoPy targets compilation rather than interpretation. This paper presents Bleach, a dynamically typed, object-oriented programming language with a tree-walk interpreter implemented in C++, designed to fill the gap left by these alternatives. Structurally inspired by Lox (from *Crafting Interpreters*), Bleach extends it with multi-line comments, do-while and for loops, elif clauses, a ternary operator, a list type, break/continue statements, an expanded standard library, and a resolver that statically detects 10 distinct classes of scoping and control-flow errors. We present the language's design rationale, describe the architecture and key implementation decisions of its interpreter, and compare Bleach feature-by-feature with ChocoPy, COOL, and MiniJava, positioning it in the space of compiler-teaching languages. We validated the interpreter through a 44-case test suite covering all AST node types and by implementing 11 standard algorithms and data structures in the language; all test cases pass, and every algorithm runs correctly without modification to the language. Despite the lack of an empirical pedagogical evaluation, we outline how Bleach could be incorporated into a 15-week undergraduate introductory Compilers course.

Keywords

Teaching, Compilers

1 Introduction

Compilers courses are a standard component of undergraduate Computer Science and Computer Engineering curricula, where they introduce students to the formal foundations and engineering of programming-language implementation. The pedagogical literature on these courses has converged, over the last three decades, on the value of project-based, hands-on teaching: Students implement either a compiler or an interpreter for a small language as the spine of the course, with lectures building up the theoretical content alongside the implementation work [1, 10, 11, 14]. The choice of project language matters. It determines the difficulty curve students experience, the topics the course can naturally cover, and the burden on the teaching staff who must provide a specification, a reference implementation, and supporting infrastructure for the language students will be working on [2].

Several languages have been designed specifically to serve as such a project artifact. COOL [2] and MiniJava [3] are deliberately minimal and statically typed. ChocoPy [16] adds richer language constructs and a formal specification but targets compilation to RISC-V assembly instructions rather than interpretation. None of these languages occupies the intersection of *dynamically typed*, *interpreter-based*, and *feature-rich enough to support object-oriented programs with closures and first-class functions*. This intersection is not idle: It aligns with a particular pedagogical stance, that is, students learn language implementation more effectively by first working through an interpreter for a relatively familiar dynamically typed language [11], focusing on the semantic and runtime concerns, before encountering the additional complexity of static type-checking and code generation. Instructors who hold this stance currently have two options: build their own language and interpreter, a multi-semester effort that exceeds what most teaching staff can sustain alongside other duties [2], or adapt a language designed for a different stance and accept the resulting friction. Neither option is satisfactory.

This paper presents Bleach, a programming language and tree-walk interpreter (implemented in C++) designed to fill in the missing design point. Bleach is dynamically typed, supports object-oriented programming with single inheritance, treats functions as first-class values, includes anonymous functions, and ships with a standard library covering arithmetic, I/O, time, randomness, and string utilities. Structurally, Bleach is descended from Lox, the language used throughout Nystrom's *Crafting Interpreters* [14]. Bleach inherits Lox's tree-walk interpreter strategy and extends Lox with multi-line comments, `do-while` and `for` loops, `elif` clauses, a ternary operator, a list type, `break` and `continue` statements, a more substantial standard library, and a resolver that detects a broader range of static errors. The interpreter is organized so that its phases (lexing, parsing, resolving, evaluating) map onto separate components that can be introduced incrementally over a 15-week course, with each week's lecture adding one piece on top of the previous week's working code. The scope of both the language and the course is lexing, parsing, static resolution, and interpretation; code generation for a physical or virtual machine, and static type checking, are deliberately left to a follow-on course, a design choice we return to in Section 7.

Three lines of evidence support the claim that Bleach is fit for its intended use. First, a 44-case test suite exercises every type of AST node and a representative selection of standard-library functions, verifying that the interpreter behaves as specified. Second, standard algorithms and data structures from the undergraduate Computer Science curriculum have been implemented in Bleach: binary search, sorting algorithms, breadth-first and depth-first search, Dijkstra's shortest-path algorithm, queues, stacks, and binary search trees. This demonstrates that the language is expressive enough for the kinds of programs students typically encounter in coursework. Third, a feature-level comparison with ChocoPy, COOL, and MiniJava (Section 3) confirms the design-point claim:

no prior educational programming language combines dynamic typing, interpreter-based execution, closures, and first-class functions. Empirical evaluation of Bleach in a classroom setting, which constitutes the most direct form of evidence for the pedagogical claims, is out of scope for this paper.

The remainder of the paper is organized as follows. Section 2 surveys prior educational programming languages and the practice-oriented teaching literature on which Bleach builds. Section 3 presents the design of the Bleach language. Section 4 describes the interpreter implementation, focusing on the resolver and runtime. Section 5 outlines how Bleach fits into a 15-week undergraduate Compilers course. Section 6 reports the validation we performed. Section 7 concludes and discusses future work.

2 Educational Programming Languages For Compilers Courses

Approaches to teaching Compilers have evolved from theory-heavy lectures toward project-based courses with language implementation work [1]. Lasseter [11] reported that students at Hobart & William Smith Colleges struggled most with semantic analysis and code generation when implementing a compiler, and that introducing the structure of an interpreter as a conceptual scaffold materially improved comprehension of those later phases. Kundra and Sureka [10] reported similar gains from a case-based and project-based methodology applied to a Compilers course in India. Nystrom [14] extends this trajectory: *Crafting Interpreters* is a book-length, project-based introduction to language implementation, structured around two interpreters for a single language (Lox): one tree-walk, one bytecode.

A related but distinct tradition uses languages such as Racket [5] and Pyret [12] as the *host* language in which students build interpreters for smaller languages. Racket in particular shares several features with Bleach: it is dynamically typed, supports closures and first-class functions, and is itself interpreter-based. The distinction we draw is one of role rather than of feature overlap: in the Racket-based tradition, the language is the tool used to implement other languages, and the interpreter under construction is typically small and changes from assignment to assignment; in the tradition Bleach belongs to, the language is itself the fixed *target* of a single, incrementally extended interpreter built over an entire semester. Both traditions are legitimate paths into language implementation; Table 1 and the languages discussed below concern only the second role, where Racket and Pyret would not be a natural fit.

COOL (Classroom Object-Oriented Language) is a small, object-oriented, statically typed language developed by Aiken at Stanford in the 1990s [2]. It includes a fixed set of base types (`Int`, `Bool`, `String`, `IO`, `Object`), single inheritance, and automatic garbage collection. Aiken's stated motivation [2] was that full-scale languages place an unsustainable burden on teaching staff, who must produce detailed specifications, supporting software, and a reference implementation before the course can use a language as its project target.

C* (C Star) is a programming language created in 2017 by Kirsch [9] as a component of the Selfie Open-Source Software Project [8, 15]. Kirsch's stated motivation was to teach the basics of language implementation to larger audiences beyond Computer Science majors. C* recognizes six keywords (`int`, `while`, `if`, `else`, `return`, `void`) and two data types (signed integer `int` and pointer to signed integer `int*`). Functions support parameters, local variables, and a return value; statements cover the usual arithmetic operations, assignment, conditionals, while loops, function calls, and return. The language is deliberately low-level and procedural, in contrast to the object-oriented designs above.

MiniJava is a subset of Java introduced by Appel and Palsberg [3]. It retains classes, methods, single inheritance, conditionals, and `while` loops, but omits exceptions, generics, interfaces, and lambdas, with the goal of letting students focus on the fundamentals of compiler implementation without the full complexity of Java.

ChocoPy is a statically typed subset of Python 3.6 with explicit type annotations, designed by Padhye et al. [16] for the Compilers course at UC Berkeley. ChocoPy is formally specified (grammar, typing rules, and operational semantics) and student projects target the 32-bit RISC-V instruction set. The language supports user-defined classes, lists, and conditionals. It omits first-class functions, the `dict` type, and reflective introspection.

Of the four languages above, all of them are statically typed and compilation-oriented. None combines dynamic typing, interpreter-based execution, and a feature set rich enough for object-oriented programs with closures and first-class functions. Bleach occupies this position in the design space. Structurally, Bleach descends from Lox, as it inherits Lox's overall organization and tree-walk interpreter strategy, and extends it with multi-line comments, `do-while` and `for` loops, `elif` clauses, a ternary operator, a list type, `break` and `continue` statements, an expanded standard library, and a resolver that detects a broader range of static errors.

3 The Bleach Language

Bleach is a high-level, dynamically typed, object-oriented programming language with closures, first-class functions, and a small standard library. Programs are organized as sequences of statements, optionally grouped into top-level functions and classes. The interpreter offers two operating modes: a REPL for short interactive sessions and a file-based mode that executes `.bch` source files. Listing 1 illustrates the language through a small object-oriented program that exercises classes, single inheritance, method overriding, the `self` and `super` keywords, and a call into the `std::math` standard library namespace. The remainder of this section enumerates the language's core features, isolates the four design decisions that distinguish Bleach from prior educational alternatives, and concludes with a feature comparison that makes the design-point claim concrete.

Listing 1: A Bleach program demonstrating classes, single inheritance, and method overriding.

```
class Shape {
  method init(name) { self.name = name; }
  method area()     { return 0; }  // overridden
}

class Circle inherits Shape {
  method init(radius) {
    super.init("Circle");
    self.radius = radius;
  }
  method area() {
    return std::math::pow(self.radius, 2) * 3.14159;
```

```
  }
}

let c = Circle(5);
print c.name;      // "Circle"
print c.area();    // 78.5398...
```

3.1 Core Features

Data types and operators: Bleach provides five built-in types: `bool`, `nil`, `num`, `str`, and `list`. The `nil` value denotes the absence of a value, in the sense of Python's `None` [6] rather than a null pointer: `nil` is a first-class value that can be stored, compared, and passed like any other, and operations that would dereference a missing value (e.g., calling a non-callable value, or accessing an undefined property) raise a runtime error rather than causing undefined behavior. `nil` occupies its own type rather than acting as a distinguished member of every other type, so Bleach has no implicit union or nullable-type machinery beyond ordinary dynamic typing. The `num` type unifies integers and floating-point numbers under a double-precision representation. The `list` type is dynamically sized and may hold values of mixed types; lists and strings expose methods (`append`, `getAt`, `setAt`, `size`, `find`, `substr`, and others) accessed via dot notation. Bleach supports the usual arithmetic, comparison, equality, and logical operators, along with a ternary conditional. Bleach adopts Ruby's truthy/falsey convention [13], in which `false` and `nil` are the only falsey values; any other value evaluates as true in boolean contexts.

Control flow: Bleach supports the standard conditional construct (`if`/`elif`/`else`) and three loop forms (`while`, `do-while`, and a C-style three-part `for`). `elif`, `do-while`, and `for` all extend Lox, which provides only `if`/`else` and `while`. Jump statements include `break`, `continue`, and `return`. The body of a conditional or loop must be a block delimited by curly braces; a non-block body is a syntax error. Each block introduces a new lexical scope, which the resolver tracks for static error detection (Section 4).

Functions and closures: Top-level functions are declared with the `function` keyword. Bleach treats functions as first-class values: they may be assigned to variables, passed as arguments, returned from other functions, and stored in data structures. Anonymous functions are written with the `lambda` keyword and an arrow syntax (`lambda -> (a, b) { return a + b; }`) and are full closures over the enclosing lexical scope. Functions that omit an explicit `return` return `nil` by default. Bleach does not support optional parameters, default values, or function overloading; each call site must match the declared arity. Listing 2 illustrates closures and first-class functions through a counter-generating function.

Listing 2: A Bleach program demonstrating closures and first-class functions.

```
function makeCounter() {
  let count = 0;
  return lambda -> () {
    count = count + 1;
    return count;
  };
}

let c = makeCounter();
print c();  // 1
print c();  // 2
print c();  // 3
```

Object-oriented model: Classes are declared with the `class` keyword and contain methods declared with the `method` keyword. As Bleach is dynamically typed, methods do not require explicit type annotations on parameters or return values; argument types are checked dynamically when methods are invoked. Instances are created by invoking the class name as a function (e.g., `let c = Circle(5)`); if a special `init` method is defined, it functions as the constructor. The `self` keyword inside a method refers to the receiving instance. Bleach supports single inheritance via the `inherits` keyword, with method overriding and access to overridden methods through the `super` keyword. As in Lox [14] and Python [6], instance attributes may be added dynamically at runtime through assignment to undeclared fields.

3.2 Design Decisions

The design space of educational programming languages is dense, and many of Bleach's individual features are unremarkable in isolation. The following four decisions, however, jointly characterize Bleach's position in the space and warrant explicit justification.

Dynamic typing: Bleach is dynamically typed. Variables hold values of any type, and type errors surface at runtime rather than at a compile-time type-checking phase. This is the most consequential design choice differentiating Bleach from the prior educational languages discussed in Section 2, all of which are statically typed. The choice serves the pedagogical stance of Section 1: omitting static type checking removes the need for a separate type-checking phase and its associated inference algorithm, freeing course time for the parser, resolver, and runtime, the components that give the language its operational behavior. We understand that this may leave students underprepared for industry roles. However, this topic could be covered in a second course, focused on compilation-oriented languages. Furthermore, we hypothesize that our design choice also makes the resulting interpreter simpler for students to read, modify, and extend; like the paper's other pedagogical claims, this hypothesis awaits the classroom evaluation discussed in Section 7.

A single numeric type: Bleach provides one numeric type, `num`, implemented as a double-precision IEEE-754 floating-point value. This unification trades precision and performance for simplicity: Students do not need to reason about integer-vs-floating-point conversions, integer overflow, or numeric coercion when implementing arithmetic operators in the interpreter. The loss of integer-exact representation for large values is accepted. It is not the language's role to support numerical methods. The same design appears in Lox [14] and (until the introduction of BigInt) in JavaScript, and is familiar to students who have used either.

Single inheritance: Bleach supports only single inheritance. The decision was driven by the desire to keep the inheritance hierarchy and method dispatch in the runtime simple to explain and to implement. Multiple inheritance introduces the well-known Diamond Problem and complicates method resolution; in a teaching language

whose runtime students will read and extend, the additional complexity is not worth the modest expressive gain. Single inheritance is also the model most undergraduate students will already have encountered in Java or Python, which keeps the language's surface familiar.

Inclusion of `nil`*:* Bleach includes a `nil` value (just as Python's `None` value) despite Hoare's well-known characterization of null references as a "billion-dollar mistake" [7]. In a statically typed language, option types or non-nullable references can offer a principled alternative. In a dynamically typed language, attempting to exclude `nil` introduces more friction than it removes: every operation that may legitimately produce "no value", such as a missing key in a lookup, an empty list pop, a function that exits without an explicit `return`, and so on, must be given some other sentinel, and dynamic typing prevents the choice of sentinel from being enforced. Lox makes the same decision for this exact reason [14].

3.3 Comparison With Prior Educational Languages

Table 1 summarizes the comparison of Bleach features with ChocoPy, COOL, and MiniJava. First-class functions, anonymous functions, and `break`/`continue` support distinguish Bleach from *all* of the compared languages; the type-system and execution-model rows distinguish Bleach from the compiled and statically typed group. Together, they substantiate the design-point argument of Section 1: Bleach is the only entry in the table that combines dynamic typing, interpreter-based execution, closures, and first-class functions.

4 Implementation: A Tree-Walk Interpreter

4.1 Grammar

The Bleach grammar is implemented by a top-down recursive descent parser. The expression grammar uses a precedence chain of 11 levels, from the lowest precedence (assignment) to the highest (primary), with each level implemented as a parsing method that delegates to the next. The statement grammar dispatches on the leading keyword. Listing 3 shows a representative fragment; the complete grammar is available in the accompanying artifact.

Listing 3: A fragment of the Bleach grammar in Backus–Naur Form (simplified for presentation).

```
program    ::= statement* EOF
statement  ::= classDecl | funcDecl | varDecl
             | ifStmt | whileStmt | forStmt
             | block | exprStmt | ...
classDecl  ::= "class" IDENT ("inherits" IDENT)?
               "{" method* "}"
funcDecl   ::= "function" IDENT "(" params? ")" block
varDecl    ::= "let" IDENT ("=" expression)? ";"
ifStmt     ::= "if" "(" expression ")" statement
               ("elif" "(" expression ")" statement)*
               ("else" statement)?
whileStmt  ::= "while" "(" expression ")" block
block      ::= "{" statement* "}"
expression ::= assignment
assignment ::= (call ".")? IDENT "=" assignment | ternary
ternary    ::= logicalOr ("?" expression ":" expression)?
primary    ::= NUMBER | STRING | "true" | "false" | "nil"
             | "self" | "super" "." IDENT | IDENT
             | "(" expression ")"
             | "[" (expression ("," expression)*)? "]"
             | "lambda" "->" "(" params? ")" block
```

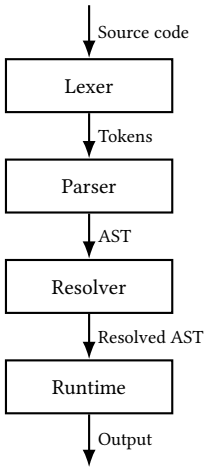

Figure 1: The four-phase pipeline of the Bleach interpreter.

4.2 Architecture

The Bleach interpreter is implemented in C++ and organized as a four-phase pipeline, illustrated in Figure 1: lexer → parser → resolver → runtime. The lexer produces a token stream from source text. The parser produces an abstract syntax tree (AST) from the token stream via recursive descent parsing. The resolver performs a second pass over the AST to record lexical-scope information and detect a set of static errors. The runtime evaluates the AST against an environment chain to produce program output. The four phases are realized as separate C++ components with explicit interfaces, allowing each to be introduced, modified, or replaced independently. The organization is itself a pedagogical choice: It maps directly onto the lecture sequence of an undergraduate Compilers course (Section 5), with each phase corresponding to one or two lectures and to a self-contained component the student can implement on top of the previous week's working code.

4.3 Lexer And Parser

The lexer is hand-written: Each token type is recognized by a small switch and per-class character predicates. Numeric literals are double-precision; string literals are double-quoted. The lexer recognizes approximately 30 token types, including reserved keywords and operators specific to Bleach (`->` for lambda, `?` and `:` for the ternary).

The parser is recursive descent. Each grammar rule maps to a method on the `Parser` class, `ifStatement`, `classDeclStatement`, `ternary`, and so on. The call structure of those methods mirrors the grammar's precedence and nesting. Error recovery is panic mode: When a parse error is raised, the parser discards tokens until it reaches a statement boundary (a semicolon or the next statement-introducing keyword) and resumes. We chose hand-written parsing over parser generators (e.g., ANTLR, Bison) because the resulting code is among the clearest direct illustrations of the parsing technique a course can present [4, 14].

Table 1: Feature-level comparison of Bleach with prior educational programming languages for Compilers courses.

Feature	Bleach	ChocoPy	COOL	MiniJava
Type system	Dynamic	Static	Static	Static
Execution model	Tree-walk interp.	Compiler	Compiler	Compiler
Paradigm	OOP	OOP	OOP	OOP
Inheritance	Single	Single	Single	Single
Top-level functions	Yes	Yes	No (methods only)	No (methods only)
First-class functions	Yes	No	No	No
Anonymous functions	Yes (`lambda`)	No	No	No
List type	Yes	Yes	No	Arrays only
Loop constructs	`while`, `do-while`, `for`	`while`, `for-in`	`while`	`while`
`break` / `continue`	Yes	No	No	No

4.4 Resolver

The resolver performs a single pass over the AST between parsing and evaluation. Its two responsibilities are (a) recording, for every variable reference, the lexical-scope distance to the variable's declaration, so the runtime can look up local variables in constant time; and (b) detecting a set of static errors that do not require type information to identify.

The scope-tracking mechanism of the interpreter is composed of a stack of `std::map<std::string, bool>`. Entering a block, function, or class pushes a new scope; exiting pops it. A variable name is first *declared* (mapped to `false`) when its declaration is encountered, then *defined* (set to `true`) only after its initializer has been resolved. This two-state pattern catches a subtle error: reading a variable inside its own initializer (e.g., `let a = a + 1`, where the right-hand `a` would, in a more permissive language, silently refer to an outer-scope `a`). When a variable reference is encountered, the resolver walks the stack from innermost to outermost, finds the declaring scope, and passes the number of stack frames between the current scope and the declaring one (distance) to the interpreter. The interpreter stores this distance in a per-expression table and uses it at evaluation time to skip directly to the correct environment, avoiding a linear walk of the environment chain.

Three enumeration variables (`currentClass`, `currentFunction`, `currentLoop`) track contextual state, allowing the resolver to detect 10 distinct static errors with no further machinery: variable redeclaration in the same scope, reading a variable in its own initializer, `self` outside a class, `super` outside a class, `super` in a class without a superclass, `break` outside a loop, `continue` outside a loop, a class inheriting from itself, `return` outside a function, and a value-bearing `return` inside an `init` method. These are scoping and control-flow-discipline errors, detectable without any type information. They are not type errors, since Bleach performs no static type analysis. Lox's resolver [14] detects most of these; the additions in Bleach are tied to the language features it adds beyond Lox, principally `break` and `continue`.

Listing 4: Sample resolver diagnostics on three ill-formed programs.

```
// Program 1: self outside a class
function f() { return self; }
//   Error: Cannot use 'self' outside of a class

// Program 2: return in init method
class Foo {
  method init() { return 42; }
}
//   Error: Cannot use the 'return' keyword inside
//   the 'init' method of a class

// Program 3: variable in own initializer
{
  let x = x + 1;
}
//   Error: Cannot read local variable in its own
     initializer
```

4.5 Runtime

The runtime is a tree-walk interpreter that implements both the `ExprVisitor` and `StmtVisitor` interfaces over the AST. Each AST node's `accept` method dispatches to a `visit*` method on the interpreter, which produces a value (for expressions) or produces side effects (for statements). Values are stored in `std::any`; type checks at operator boundaries are performed dynamically. Listing 5 shows the skeleton of the Bleach visitor-pattern dispatch. `accept`/`shared_from_this` achieve double dispatch; `std::any` is the universal value type; non-local control flow escapes as typed C++ exceptions, thrown at the statement site and caught at the enclosing loop or function boundary.

Listing 5: Skeleton of the Bleach visitor-pattern dispatch.

```
struct ExprVisitor {
  virtual std::any visitBinaryExpr(std::shared_ptr<Binary
      > expr) = 0;
  // further visit* methods (Assign, Call, Literal, ...)
  virtual ~ExprVisitor() = default;
};
// StmtVisitor...

struct Expr { virtual std::any accept(ExprVisitor&
    visitor) = 0; };

struct Binary : Expr, std::enable_shared_from_this<Binary
    > {
  Token op;  std::shared_ptr<Expr> left, right;
  std::any accept(ExprVisitor& visitor) override {
    return visitor.visitBinaryExpr(shared_from_this());
        // double dispatch
  }
};

class Interpreter : public ExprVisitor, public
    StmtVisitor {
  std::any evaluate(std::shared_ptr<Expr> e) { return e->
      accept(*this); }
  public:
```

```
  std::any visitBinaryExpr(std::shared_ptr<Binary> expr)
      override {
    std::any left = evaluate(expr->left);
    std::any right = evaluate(expr->right);
    if (checkNumberOperands(left, right))
      return std::any_cast<double>(left) + std::any_cast<
          double>(right);
  }
  std::any visitReturnStmt(std::shared_ptr<Return> stmt)
      override {
    throw BleachReturn{ stmt->value ? evaluate(stmt->
        value) : nullptr };
  }
  std::any visitBreakStmt(std::shared_ptr<Break> stmt)
      override {
    throw BleachBreak{};
  }
};

std::any BleachFunction::call(Interpreter& interp, /*...
    */) {
  try { interp.executeBlock(body, env); }
  catch (BleachReturn r) { return r.value; }
  return nullptr; // implicit nil
}
```

Scope is implemented via a chain of `Environment` objects, each holding a name-to-value map and a pointer to its enclosing environment. Local-variable access uses the distance information recorded by the resolver: The interpreter walks exactly n enclosing pointers to reach the declaring environment, where n is the resolver-supplied distance. Closures are implemented by storing the current environment in the function value at the point of declaration; when the function is invoked, a fresh local environment is created on top of the captured environment, so the function sees the enclosing variables at its declaration site rather than at its call site.

Language constructs for non-local control flow, such as `break`, `continue`, and `return`, are implemented as C++ exceptions thrown from the corresponding visit methods and caught at the appropriate enclosing loop or function boundary. This is a deliberately simple mechanism that avoids the explicit state machine and an interpreter loop that would otherwise be needed.

Method dispatch on instances uses a per-class method table populated at class-declaration time. The `super` keyword resolves via the class hierarchy: the resolver records the depth at which `super` is bound in the environment chain, and the runtime uses that depth to locate the superclass and bind the named method to the receiving instance.

4.6 Standard Library

The standard library is exposed as a flat namespace of native functions registered in the global environment at interpreter start-up. Namespaces include `std::math` (`abs`, `ceil`, `floor`, `log`, `pow`, `sqrt`), `std::io` (`readLine`, `print`, `fileRead`, `fileWrite`), `std::utils` (`ord`, `strToNum`, `strToBool`, `strToNil`), `std::random::random`, and `std::chrono::clock`.

4.7 Implementation Characteristics

The Bleach interpreter consists of more than 3,500 lines of C++17, depends only on the standard library, and builds with any conforming compiler; our reference build uses g++ 13.3.0. There is no external parser-generator, tokenizer-generator, or runtime-library dependency. The full test suite of Section 6 executes in under 0.160 seconds on a laptop computer running Ubuntu 24.04.1 LTS, equipped with an Intel Core i5-1135G7 processor (4 cores, 2.40 GHz), 8 GB of RAM, and an Intel Iris Xe integrated graphics card. Exact build instructions and dependency versions are documented in the artifact repository (see Artifact Availability, below). These characteristics matter pedagogically: Students can build and run the interpreter on their own machines without configuring a toolchain, and the codebase is small enough that they can plausibly read it end-to-end as the course progresses.

5 Bleach In An Undergraduate Compilers Course

Bleach is intended to support a course structure in which students incrementally build a tree-walk interpreter for the language over a 15-week semester, with each week's lecture introducing one component or feature, and each week's assignment extending the working interpreter to support what was lectured on. Table 2 shows the proposed structure, with each row identifying the topic lectured, the implementation work expected, and the corresponding phase or feature of Bleach exercised. This structure follows the dependency order natural to a tree-walk interpreter and broadly mirrors the chapter sequence of existing books on the subject [3, 4, 14], with weeks 13 and 14 reordered so that the resolver precedes object-oriented features.

Several properties of the structure are worth noting. First, the ordering follows the feature-dependency graph of the interpreter. Variables (week 9) are introduced after the basic interpreter (weeks 7–8) is in place. Control flow (week 10) builds on variables. Functions (week 11) build on control flow. The resolver (week 13) operates over the already-implemented AST. Object-oriented features (week 14) compose method dispatch onto the existing function machinery. Second, the language design has been calibrated so that each week's extension is local: adding `break` and `continue` in week 10 requires new AST node types and new visitor cases but does not modify earlier ones; adding the resolver in week 13 introduces a new pass while requiring only minor changes to variable lookup in the runtime. Third, several features are designed as extensions that sit above the minimum required: in week 10, the `if`/`while` core is mandatory, while `for`, `do-while`, `elif`, `break`, and `continue` are extension exercises; in week 11, the core is named functions, with anonymous functions as an extension. This separation lets instructors pace the course to the cohort, asking students who progress quickly to implement the extensions and assessing the minimum from those who do not. The artifact accompanying this paper includes a reference implementation of all features in the table.

6 Validation

We validate Bleach and its interpreter from two perspectives: correctness of the interpreter against a hand-crafted test suite, and

Table 2: A proposed 15-week structure for an undergraduate Compilers course built around incremental implementation of the Bleach tree-walk interpreter.

Weeks	Lecture topic	Concepts introduced	Implementation work
1–2	Languages and interpreters	Compilers vs. interpreters; review of regular expressions, finite automata, and CFGs; introduction to Bleach	—
3–4	Lexical analysis	Lexemes, tokens, maximal-munch	Lexer for Bleach
5–6	Syntax analysis	ASTs; top-down vs. bottom-up parsing; recursive descent	Parser for the expression subset
7–8	Tree-walk interpretation	Tree traversal; the Visitor pattern	Interpreter for literals and operators
9	Variables and scope	Variable declaration, blocks, environments, shadowing	Environment chain; variable statements
10	Control flow	`if`, `while`, and (as extensions) `for`, `do-while`, `break`, `continue`, `elif`	Control-flow statement nodes
11	Functions and closures	Function declaration, call, return; closure semantics; (extension) anonymous functions	Function values, environment capture
12	Standard library	Native functions; foreign-function interface	Selected native functions
13	Resolving	Static analysis; name resolution; pre-runtime error detection	Resolver pass
14	Object-oriented features	Classes, instances, methods, `self`, single inheritance, `super`	Class declarations and method dispatch
15	Presentation and assessment	—	Project presentation, final exam

expressiveness of the language through implementations of standard algorithms and data structures from the undergraduate Computer Science curriculum. Both bodies of evidence are reproducible from the artifact accompanying this paper. Direct evidence for the pedagogical claims, whether Bleach actually improves student outcomes in a real Compilers course, is out of scope for this paper and is discussed in Section 7.

Interpreter correctness: The test suite consists of 44 cases organized into three groups: tests that exercise expression-AST nodes, tests that exercise statement-AST nodes, and tests that exercise standard-library native functions from the `std::math` and `std::utils` namespaces. Every AST node type defined by the parser is exercised by at least one test. Each test case is a triple of files: a `.bch` program, a `.bch.expected` file containing the program's expected output, and a `.bch.log` file produced by running the interpreter on the program. A shell script automates the comparison: A test passes when the `.bch.log` matches the `.bch.expected` exactly. The suite runs in a few seconds and functions as a smoke-test gate during the development of language features, not a comprehensive correctness proof. In particular, the current suite covers positive-path behavior: programs that should produce a given output. For positive cases, the suite covers correctness; for the static error classes detected by the resolver, the current artifact verifies them manually rather than through automated tests. As an illustration, the program `class A inherits A { }` produces the diagnostic `"A class cannot inherit from itself"`, sourced from the resolver rather than from runtime execution. Negative-path coverage, that is, programs that should produce a specific diagnostic, particularly for the 10 static errors detected by the resolver (Section 4.4), is not yet included and is discussed under future work.

Language expressiveness: A separate body of Bleach code implements 11 standard algorithms and data structures: binary heap, binary search tree, binary search, breadth-first and depth-first search, Dijkstra's shortest-path algorithm, insertion sort, merge sort, queue, quicksort, and stack. These are programs of the kind students typically write in a second-year Algorithms and Data Structures course, and their expression in Bleach requires the simultaneous use of classes, methods, single inheritance for the data-structure type hierarchies, closures for callback-style code in the graph algorithms, the `list` type, recursion, and the standard library. The point is not the algorithms themselves, as their implementations are unsurprising, but that the language is rich enough to express them naturally, without workarounds for missing features. This is the property an instructor needs in order to assign non-trivial programs in the language, and it is the property that distinguishes Bleach from the more minimal designs (COOL, MiniJava) discussed in Section 2.

Threats to validity: Some limitations of this validation deserve explicit acknowledgment. First, a hand-crafted test suite reflects the implementer's mental model of the language and is biased toward the cases the implementer has thought to test; a stronger correctness story would involve property-based testing or differential testing against a reference implementation. Second, the algorithm implementations demonstrate that Bleach *can* express these programs, but not that *students* can write them comfortably; the relevant measurement is a usability or learning-outcomes study with a real cohort, which we revisit in Section 7. Finally, the comparison in Table 1 reflects features we deemed pedagogically relevant; a different selection might shift the relative positioning of the languages.

7 Conclusion and Future Work

We presented Bleach, a dynamically typed, object-oriented programming language with a tree-walk interpreter, designed to occupy a previously unfilled niche in the space of educational programming languages for undergraduate Compilers courses. Bleach descends from Lox [14] and extends it with control-flow constructs (`do-while`, `for`, `elif`, `break`, `continue`, ternary), a list type, anonymous functions, an expanded standard library, and a resolver that detects 10 distinct static errors. A feature comparison with ChocoPy, COOL, and MiniJava substantiates the design-point claim: None of these existing options combines dynamic typing, interpreter-based execution, closures, and first-class functions. We have also proposed a 15-week course structure built around incremental extension of the interpreter, calibrated so that each week's lecture corresponds to one component or feature and each week's assignment builds on the previous week's working code.

Scope: Bleach and the course structure of Section 5 are scoped to lexical analysis, parsing, static resolution, and tree-walk interpretation; neither the language nor the proposed course addresses code generation for a physical or virtual target machine, nor static type checking. We view this scoping as deliberate rather than as an omission: It targets a first course in language implementation, one that pairs naturally with a subsequent course or module built around a compilation-oriented, statically typed language such as ChocoPy or MiniJava, in which students revisit many of the same programs under a different set of constraints.

Limitations and future work: The main limitation of this paper is the absence of direct empirical evidence for Bleach's pedagogical value: the language has not yet been used in a real Compilers course, and the proposed course structure of Section 5 has not yet been taught. A classroom study is the central next step. Concretely, we plan a pilot deployment in an undergraduate Compilers course, with a pre-course survey of student backgrounds, weekly assignment-completion tracking, an end-of-course usability and learning-outcomes survey, and, where feasible, a comparison condition against a cohort using traditional materials. Complementing the classroom study, the test-suite limitations identified in Section 6 motivate adding negative-path coverage (programs that should produce specific resolver diagnostics) and exploring property-based or differential testing against a reference implementation.

Beyond evaluation, several language and implementation extensions would broaden Bleach's pedagogical reach. List and string indexing syntax (currently expressed through the `getAt` and `setAt` methods) is the most user-visible gap and would provide a natural exercise in parsing and runtime checks. A `const` keyword for constant bindings would give students a concrete reason to revisit the resolver and environment design. A `for...in` loop, as a piece of syntactic sugar desugared at parse time, would introduce the notion of source-level transformation as a course topic in its own right. The resolver itself could be extended to detect additional static errors (unused variables, unreachable code, function redeclaration), each addition giving the course a self-contained analysis-pass exercise.

Artifact Availability

The Bleach reference implementation is publicly available at https://github.com/vmmc2/Bleach and archived on Zenodo at https://doi.org/10.5281/zenodo.21713832. The repository includes the C++ tree-walk interpreter, build, run, and clean scripts, the 44-case test suite discussed in Section 6 together with the shell script that automates it, the eleven algorithm and data-structure programs used to assess language expressiveness, and the language's context-free grammar. A companion documentation site and a syntax-highlighting extension for Visual Studio Code are also linked from the repository.

Acknowledgments

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence, www.ines.org.br), under CNPq grant 408817/2024-0. Leopoldo is partially supported by CNPq grant 310405/2025-4.

References

[1] Alfred V. Aho. 2008. Teaching the compilers course. *ACM SIGCSE Bulletin* 40, 4 (2008), 6–8.
[2] Alexander Aiken. 1996. Cool: A portable project for teaching compiler construction. *ACM Sigplan Notices* 31, 7 (1996), 19–24.
[3] Andrew W. Appel and Jens Palsberg. 2002. *Modern Compiler Implementation in Java, 2nd Edition.* Cambridge University Press.
[4] Keith D Cooper and Linda Torczon. 2022. *Engineering a compiler.* Morgan Kaufmann.
[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing* (second ed.). MIT Press.
[6] Python Software Foundation. 2024. Python Programming Language, Version 3.12.5. https://www.python.org/. Accessed: 2024-09-02.
[7] Tony Hoare. 2009. Null References: The Billion Dollar Mistake. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ Presentation at QCon London 2009. Accessed 20 September 2024..
[8] Christoph Kirsch. 2017. Selfie | Official GitHub Repository. https://github.com/cksystemsteaching/selfie. Accessed: 2024-09-07.
[9] Christoph M. Kirsch. 2017. Selfie and the Basics. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* 198–213.
[10] Divya Kundra and Ashish Sureka. 2016. An experience report on teaching compiler design concepts using case-based and project-based learning approaches. In *2016 IEEE Eighth International Conference on Technology for Education (T4E).* IEEE, 216–219.
[11] John HE Lasseter. 2015. The Interpreter In An Undergraduate Compilers Course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education.* 168–173.
[12] Benjamin S. Lerner, Joe Gibbs Politz, and Shriram Krishnamurthi. 2017. Pyret. https://pyret.org/index.html. Accessed: 2024-09-02.
[13] Yukihiro Matsumoto. 2023. Ruby Programming Language. https://www.ruby-lang.org/en/ Version 3.2.
[14] Robert Nystrom. 2021. *Crafting interpreters.* Genever Benning.
[15] University of Salzburg. 2017. Selfie | An educational software system of a tiny self-compiling C compiler, a tiny self-executing RISC-V emulator, and a tiny self-hosting RISC-V hypervisor. http://selfie.cs.uni-salzburg.at/. Accessed: 2024-09-07.
[16] Rohan Padhye, Koushik Sen, and Paul N Hilfinger. 2019. Chocopy: A programming language for compilers courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E.* 41–45.