Volume 3, Issue 9, September - 2025 **ISSN (E):** 2938-3811

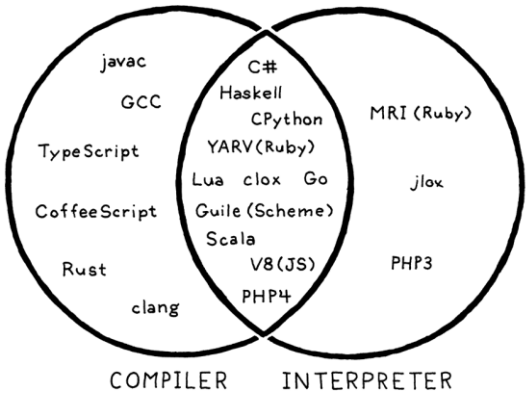
IMPLEMENTATION AND ANALYSIS OF A TREE-WALK INTERPRETER IN CPYTHON: A STUDENT'S PERSPECTIVE ON LANGUAGE DESIGN

Abduaziz Ziyodov
Author, Senior Student at Inha University in Tashkent
E-Mail: mail@ziyodov.uz
Project's Source Code: https://github.com/AbduazizZiyodov/gustav
Website: abduaziz.ziyodov.uz

Abstract
The tree-walk interpreter Gustav, created as a thorough learning exercise for programming language implementation, is presented in this paper. The project expands the Lox language with more advanced features like lambda expressions, pipe operators, ternary expressions, and improved loop constructs, building on the fundamental ideas from Crafting Interpreters[1]. The implementation serves as a basic programming language and a teaching tool for compiler design concepts, showcasing the full interpreter pipeline from lexical analysis to runtime execution. This work offers insights into language design trade-offs, implementation difficulties, and the connection between language features and their underlying computational models by methodically analyzing each compilation phase. The interpreter maintains code clarity and extensibility while achieving 100% test coverage and exhibiting typical tree-walk performance characteristics.

Keywords: Interpreter design, tree-walk, cpython.

Introduction



One of the best learning opportunities in computer science education is the implementation of programming languages, which calls for the fusion of theoretical ideas with real-world software engineering. By filling the gap between abstract language features and their concrete execution

Web of Scientists and Scholars: Journal of Multidisciplinary Research

models, interpreters' design and implementation offer direct insight into how high-level programming constructs translate into computational behavior [2].

Because of their conceptual clarity and clear alignment between source code structure and execution flow, tree-walk interpreters provide an excellent approach to language implementation [3]. Tree-walk interpreters evaluate abstract syntax trees directly, in contrast to bytecode virtual machines or native code compilers, so developers can see the connection between syntax and semantics immediately.

This paper documents the development of **Gustav**, a dynamically-typed programming language interpreter implemented in CPython. The project began as an implementation of the Lox language specification from *Crafting Interpreters* but evolved to include several language extensions that provided additional learning opportunities and implementation challenges. The **complete source** code and documentation are available at https://github.com/AbduazizZiyodov/gustav.

To demonstrate Gustav's capabilities, consider this binary search tree (BST algorithm) implementation showcasing minimal syntax of gustav, object-oriented programming, recursive algorithms, and functional composition:

```
class Tree {
   init(value) { this.value = value; this.left = this.right = nil; }
   insert(value) {
      if (value < this.value) {
         this.left = this.left == nil ? Tree(value) :
               this.left.insert(value);
      } else {
         this.right = this.right == nil ? Tree(value) :
               this.right.insert(value);
      }
   }
}

fun binary_search(node, value) {
   return node == nil ? false : node.value == value ? true :
         value < node.value ? binary_search(node.left, value) :
         binary_search(node.right, value);
}

var tree = Tree(10);

tree.insert(5); tree.insert(15); tree.insert(3); tree.insert(7);

print binary_search(tree, 7);   // true
print binary_search(tree, 15);  // true
print binary_search(tree, 12);  // false
```

This example demonstrates Gustav's support for classes with constructors, method chaining, ternary operators, recursive function calls, and clean syntax that balances readability with expressiveness.

Web of Scientists and Scholars: Journal of Multidisciplinary Research

1.1 Theoretical Background

Programming language implementation typically follows a multi-phase architecture consisting of lexical analysis (scanning), syntactic analysis (parsing), semantic analysis, and code generation or interpretation[4]. Each phase transforms the program representation, progressively refining it from character sequences to executable operations.

Lexical Analysis converts character streams into tokens, implementing finite automata to recognize language constructs. This phase handles keywords, operators, literals, and identifiers while managing whitespace and comments[5].

Syntactic Analysis constructs abstract syntax trees (ASTs) from token sequences using parsing algorithms such as recursive descent or shift-reduce techniques. The parser enforces grammatical rules and operator precedence while detecting syntax errors[6].

Semantic Analysis performs static analysis to detect semantic errors and optimize runtime performance. This phase typically includes type checking, variable resolution, and scope analysis[7].

Interpretation directly executes the AST using the Visitor pattern or similar traversal techniques, evaluating expressions and executing statements according to the language's operational semantics[8].

2. Implementation Architecture

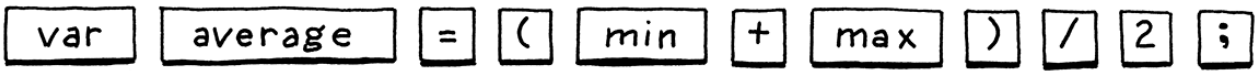
2.1 Overall Design

Gustav follows the four-phase interpreter architecture established by Nystrom[1], with each component implementing a specific interface and maintaining clear separation of concerns. The implementation emphasizes type safety through comprehensive CPython type annotations and maintains 100% test coverage across all components.

The interpreter pipeline processes source code through the following stages:

1. **Scanner** (gustav/scanner.py) - Lexical analysis producing token streams
2. **Parser** (gustav/parser.py) - Recursive descent parsing generating ASTs
3. **Resolver** (gustav/resolver.py) - Static semantic analysis and variable resolution
4. **Interpreter** (gustav/interpreter.py) - Tree-walk evaluation with runtime error handling

2.2 Lexical Analysis Implementation



The scanner implements character-by-character processing using a finite state machine approach. The core tokenization logic demonstrates how multi-character operators require lookahead processing:

```
def next_token(self) -> None:
    char: str = self.move_current()
    match char:
        case "!":
            self.add_token(type=(TT.BANG, TT.BANG_EQUAL)[self.match_token("=")])
        case "=":
            self.add_token(type=(TT.EQUAL, TT.EQUAL_EQUAL)[self.match_token("=")])
        case "|":
            if self.match_token(">"):
                self.add_token(TT.PIPE)
            else:
                gustav.panic(self.line, "Unexpected character")
```

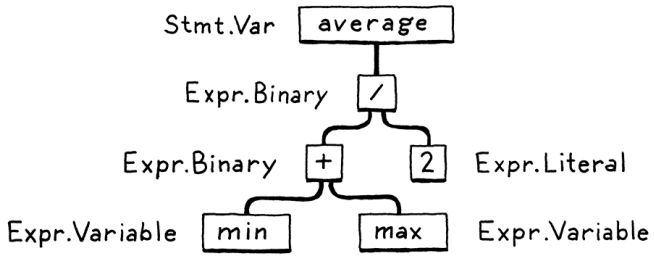
This implementation pattern allows the scanner to recognize compound operators like !=, ==, and |> while maintaining linear time complexity. The token recognition uses CPython's structural pattern matching for clarity, though traditional conditional logic would achieve equivalent performance.

Critical Implementation Detail:
Token identity management required careful consideration of CPython's dataclass equality semantics. Initially, identical tokens from different AST locations were treated as equal due to structural equality, causing variable resolution conflicts. The solution involved disabling automatic equality generation:

```
@dataclass(frozen=True, slots=True, eq=False)
class Token:
    type: TT
    lexeme: str
    literal: t.Any
    line: int
```

This ensures identity-based rather than structural equality, preventing resolution mapping collisions between syntactically identical but semantically distinct token instances.

2.3 Recursive Descent Parsing



The parser implements a recursive descent strategy with explicit precedence handling. Each grammar rule **corresponds to a parsing method**, maintaining clear correspondence between grammar specification and implementation:

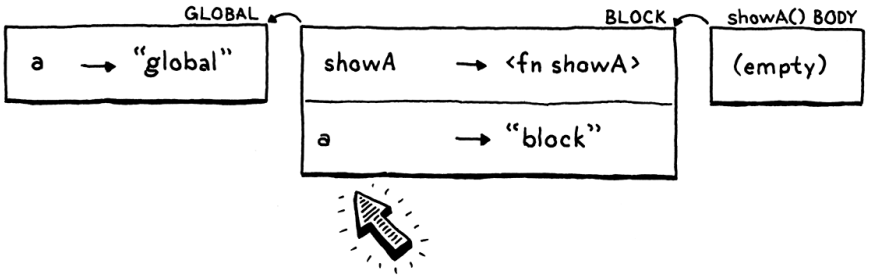
```
def parse_expression(self) -> E.Expression:
    assignment_expr = self.parse_assignment()

    if self.match(TT.PIPE):
        return self.parse_call(assignment_expr)

    return assignment_expr
```

Pipe Operator Implementation: The pipe operator (|>) demonstrates parse-time syntactic transformation. Rather than creating dedicated AST nodes, pipes are transformed into function calls during parsing:

```
def finish_call(
            self,
            callee: E.Expression,
            pipe_arg: E.Expression | None = None
) -> E.Call:
    arguments: list[E.Expression] = []

    # Parse regular arguments
    if not self.check(TT.RIGHT_PAREN):
        arguments.append(self.parse_expression())
        while self.match(TT.COMMA):
            arguments.append(self.parse_expression())

    # Append piped argument as final parameter
    if pipe_arg:
        arguments.append(pipe_arg)  # f(x) |> g(y) becomes g(y, f(x))

    paren = self.consume(TT.RIGHT_PAREN, "Expect ')' after arguments")
    return E.Call(callee, paren, arguments)
```

This transformation strategy eliminates runtime overhead while providing convenient syntax. The expression f(x)|>g(y) becomes g(y,f(x)) during parsing, requiring no special interpreter support.

2.4 Semantic Analysis Through Variable Resolution



The resolver performs single-pass static analysis to optimize variable access and detect scope-related errors. Variable resolution maps each variable reference to its declaration scope, enabling constant-time variable lookup during interpretation:

Web of Scientists and Scholars: Journal of Multidisciplinary Research

```
def resolve_local(
            self,
            expression: E.Expression,
            name: Token,
            mark_as_used: bool
) -> None:
   for i in range(len(self.scopes) - 1, -1, -1):
      if name.lexeme not in self.scopes[i]:
         continue

      depth = len(self.scopes) - 1 - i
      self.interpreter.resolve(expression, depth)

      if mark_as_used:
         self.scopes[i][name.lexeme] = (name, VariableState.USED)

      return
```

The resolver maintains a scope stack during AST traversal, computing the lexical distance between variable references and their declarations. This analysis eliminates the need for runtime scope chain traversal, improving interpreter performance while enabling early error detection.

2.4.1 Variable State Tracking

Gustav implements variable state management to catch common programming errors:

```
class VariableState(StrEnum):
   USED = auto()
   DECLARED = auto()
   DEFINED = auto()
```

Variables progress through three states: DECLARED (name reserved), DEFINED (initialized), and USED (referenced). This enables detection of several error categories:

Uninitialized Variable Access: Variables declared but not initialized cannot be used:

```
def visit_variable_expression(self, expression: E.Variable) -> None:
   if (self.has_active_scope()
      and expression.name.lexeme in self.peek()
      and (scope_value := self.peek().get(expression.name.lexeme)) is not None
      and scope_value[1] == VariableState.DECLARED):
      gustav.error(expression.name,
            "Can't read local variables in its own initializer"): ...
```

This prevents accessing variables within their own initialization expressions.

Unused Variable Warnings:

The resolver tracks variable usage and warns about unused declarations:

```
def end_scope(self) -> None:
   scope = self.scopes.pop()

   for _, value in scope.items():
      name, status = value
```


Web of Scientists and Scholars: Journal of Multidisciplinary Research

```
    if status == VariableState.DEFINED:
        gustav.warning(name, f"Variable '{name.lexeme}' is not used")
```

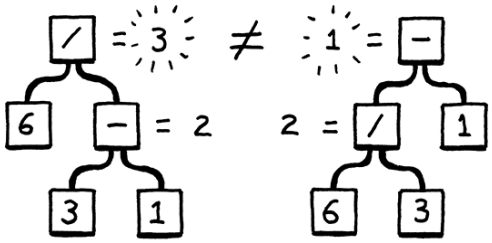
This analysis helps identify potential bugs and encourages cleaner code by flagging unnecessary variable declarations.

Runtime Uninitialized Detection:
The interpreter prevents access to uninitialized variables:

```
def visit_variable_expression(self, expression: E.Variable) -> t.Any:
    value = self.look_up_variable(expression.name, expression)

    if value is UNINITIALIZED:
        raise GusRuntimeError(expression.name,
                f"Can't use uninitialized variable'{expression.name.lexeme}'")

    return value
```

Where UNINITIALIZED is a sentinel value distinguishing declared but uninitialized variables from undefined ones.

2.5 AST Design and Tree-Walk Evaluation



Gustav implements AST nodes using CPython dataclasses with the Visitor pattern for evaluation. This design provides type safety while maintaining extensibility:

```
@dataclass(frozen=True, slots=True, eq=False)
class Binary(Expression):
    left: Expression
    operator: Token
    right: Expression

    def accept[T](self, visitor: "ExpressionVisitor[T]") -> T:
        return visitor.visit_binary_expression(self)
```

Expression Evaluation:
The interpreter implements expression evaluation through visitor methods, handling type coercion and runtime error detection:

```
def visit_binary_expression(self, expression: E.Binary) -> t.Any:
    left = self.evaluate(expression.left)
    right = self.evaluate(expression.right)

    if expression.operator.type in Interpreter.NUMERIC_OPERATORS:
        self.check_number_operands(expression.operator, left, right)
```

```
    match expression.operator.type:
    case TT.PLUS:
      if any(isinstance(left, T) and isinstance(right, T)
           for T in (str, int, float)):
        return left + right
      raise GusRuntimeError(expression.operator,
                  "Operands must be two numbers or two strings")
    case TT.STAR:
      return left * right
    case TT.SLASH:
      return float("nan") if right == 0 else left / right
```

This implementation demonstrates runtime type checking for dynamic languages and the handling of edge cases like division by zero.

3. Advanced Feature Implementation

3.1 Lambda Expressions and Closures

Lambda expressions in Gustav provide first-class function capabilities with complete closure support, implementing lexical scoping semantics that capture variable bindings at declaration time rather than execution time.

3.1.1 Theoretical Foundation of Closures

A closure consists of a function definition paired with the lexical environment in which it was defined[13]. This environment must persist beyond the original scope's lifetime, requiring careful memory management and variable capture strategies. The implementation must address several critical challenges:

1. **Variable Capture**: Which variables from the enclosing scope should be captured ?
2. **Lifetime Management**: How long should captured variables remain accessible ?
3. **Scoping Rules**: How do captured variables interact with parameters and local variables ?

3.1.2 Lambda AST Representation

Gustav represents lambda expressions as AST nodes containing parameter lists and statement bodies, similar to regular functions but without named identifiers:

```
@dataclass(frozen=True, slots=True, eq=False)
class Lambda(Expression):
  params: list[Token]
  body: list[Statement]

  def accept[T](self, visitor: “ExpressionVisitor[T]”) -> T:
    return visitor.visit_lambda_expression(self)
```

This representation allows lambdas to be treated as expressions, enabling their use in contexts where regular function declarations would be syntactically invalid.

3.1.3 Closure Capture Implementation

The lambda evaluation process creates a GusLambda callable that captures the current environment:

```
@dataclass(slots=True, frozen=True, eq=False)
class GusLambda(GusCallable):
    declaration: E.Lambda
    closure: Environment  # Captured at declaration time

    def call(self, interpreter: CanExecuteBlock, arguments: list[t.Any]) -> t.Any:
        # Create new environment with closure as parent
        environment: Environment = Environment(self.closure)

        # Bind parameters to arguments
        for I in range(self.arity()):
            environment.define(self.declaration.params[i].lexeme, arguments[i])

        try:
            interpreter.execute_block(self.declaration.body, environment)
        except GusReturn as exc:
            return exc.value

        return None

    def arity(self) -> int:
        return len(self.declaration.params)
```

Critical Design Decision:

The closure field captures a reference to the entire environment chain at declaration time. This ensures that all accessible variables remain available during lambda execution, regardless of when or where the lambda is called.

3.1.4 Environment Chain Management

The Environment class implements lexical scoping through a linked chain of variable bindings:

```
class Environment:
    def __init__(self, enclosing: "Environment | None" = None) -> None:
        self.values: dict[str, t.Any] = dict()
        self.enclosing: "Environment | None" = enclosing

    def get(self, name: Token) -> t.Any | t.NoReturn:
        if name.lexeme in self.values:
            return self.values.get(name.lexeme)

        if self.enclosing is not None:
            return self.enclosing.get(name)

        raise GusRuntimeError(name, f"Undefined variable '{name.lexeme}'")
```

When a lambda captures its environment, it maintains a reference to the entire chain, preserving access to variables from all enclosing scopes.

3.1.5 Practical Closure Example

Consider this Gustav code demonstrating closure behavior:

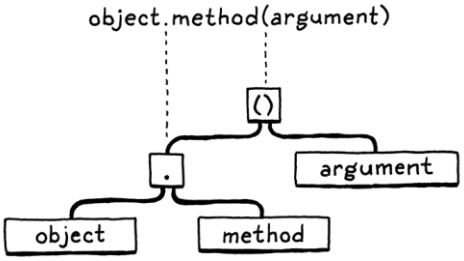
```
fun make_counter() {
  var count = 0;
  return λ() {
    count = count + 1;
    return count;
  };
}
var counter = make_counter();
print counter(); // 1
print counter(); // 2
```

The lambda captures the count variable from make_counter's environment. Even after make counter returns, the lambda retains access to count, demonstrating proper closure semantics.

3.2 Object-Oriented Programming Support

Gustav implements a complete object-oriented programming system with classes, inheritance, method dispatch, and instance management. The implementation demonstrates how OOP features can be layered onto a functional foundation.

3.2.1 Class Declaration and Instantiation



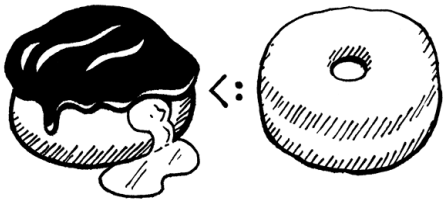
Classes in Gustav are first-class objects that can be called to create instances:

```
@dataclass
class GusClass(GusCallable):
  name: str
  superclass: "GusClass | None"
  methods: dict[str, GusFunction]

  def call(self, interpreter: CanExecuteBlock, arguments: list[t.Any]) -> t.Any:
    # Create new instance
    instance = GusClassInstance(self)

    # Call initializer if present
    if self.initializer is not None:
      self.initializer.bind(instance).call(interpreter, arguments)

    return instance

  @property
```

```
    def initializer(self) -> GusFunction | None:
        return self.find_method("init")

    def arity(self) -> int:
        return 0 if self.initializer is None else self.initializer.arity()
```

Instance Creation Process: Class instantiation follows a two-phase process: first creating an empty instance, then invoking the initializer with the instance bound to `this`. This design separates object allocation from initialization, enabling more sophisticated construction patterns.

3.2.2 Method Resolution and Inheritance



Method resolution implements single inheritance with linear search up the inheritance chain:

```
def find_method(self, name: str) -> GusFunction | None:
    # Check current class methods
    if name in self.methods:
        return self.methods.get(name)

    # Recursively search superclasses
    if self.superclass is not None:
        return self.superclass.find_method(name)

    return None
```

This implementation provides O(d) method lookup where d is the inheritance depth. While not optimal for deep hierarchies, it maintains simplicity and matches the performance characteristics of other tree-walk operations.

3.2.3 Instance Management and Field Access

Instance objects manage both fields and method access through a unified interface:

```
@dataclass
class GusClassInstance:
    klass: GusClass
    fields: dict[str, t.Any] = field(default_factory=dict)

    def get(self, name: Token) -> GusFunction | t.Any:
        # Check instance fields first
        if name.lexeme in self.fields:
            return self.fields.get(name.lexeme)

        # Look for methods in class hierarchy
```

```
        method: GusFunction | None = self.klass.find_method(name.lexeme)
        if method is not None:
            return method.bind(self)  # Bind method to this instance

        raise GusRuntimeError(name, f"Undefined property '{name.lexeme}'")

    def set(self, name: Token, value: t.Any) -> None:
        self.fields[name.lexeme] = value
```

Method Binding Mechanism: When accessing a method, the instance returns a bound version that has this pre-configured in its closure. This ensures that method calls have access to the correct instance context.

3.2.4 Method Binding Implementation

Method binding creates a new function with `this` defined in its closure:

```
def bind(self, instance: t.Any) -> "GusFunction":
    # Create new environment with this instance
    environment: Environment = Environment(self.closure)
    environment.define("this", instance)
    return GusFunction(self.declaration, environment, self.is_initializer)
```

This binding occurs at property access time, not method definition time, enabling methods to be shared across instances while maintaining per-instance context.

3.3 Enhanced Control Flow Implementation

Gustav extends basic control flow with loop-specific constructs and sophisticated break/continue handling that maintains proper environment cleanup.

3.3.1 Exception-Based Control Transfer

Break and continue statements use CPython's exception mechanism for non-local control transfer:

```
class GusStopIteration(RuntimeError):
    """Used in break statement"""
    pass

class GusContinueIteration(RuntimeError):
    """Used in continue statement"""
    pass
```

This approach provides efficient control transfer while leveraging CPython's existing exception handling infrastructure. The exceptions carry no data, serving purely as control signals.

3.3.2 Loop Environment Management

Loop statements create isolated environments to ensure proper variable scoping and cleanup:

```
def visit_for_statement(self, statement: S.For) -> None:
    # Create isolated loop environment
```


Web of Scientists and Scholars: Journal of Multidisciplinary Research

```
    loop_environment = Environment(self.environment)
    previous_environment = self.environment
    self.environment = loop_environment

    try:
        # Initialize loop variable in loop scope
        if statement.initializer:
            self.execute(statement.initializer)

        while self.is_truthy(self.evaluate(statement.condition)):
            try:
                self.execute(statement.body)
            except GusStopIteration:
                break  # Clean break from loop
            except GusContinueIteration:
                # Execute increment before continuing
                if statement.increment:
                    self.evaluate(statement.increment.expression)
                continue

            # Normal increment execution
            if statement.increment:
                self.evaluate(statement.increment.expression)

    finally:
        # Always restore previous environment
        self.environment = previous_environment
```

Environment Isolation: Each loop creates its own environment scope, ensuring that loop variables don't leak into the enclosing scope while still allowing access to outer variables.

3.3.3 Loop Statement: desugaring into "while" statement

The `loop` statement demonstrates how complex constructs can be implemented through desugaring:

```
def parse_loop_statement(self) -> S.While:
    """Desugar 'loop { ... }' into 'while (true) { ... }'"""
    condition = E.Literal(value=True)
    body: S.Statement = self.parse_statement()

    return S.While(condition, body=body)
```

This transformation occurs during parsing, eliminating the need for separate loop handling in the interpreter. The approach demonstrates how syntactic convenience can be provided without runtime complexity.

3.3.4 Nested Loop Break/Continue "guards"

The resolver ensures that break and continue statements only appear within loop contexts:

```
def visit_break_statement(self, statement: S.Break) -> None:
    if not self.in_loop:
```

```
    gustav.error(statement.keyword, "Can't use 'break' outside of a loop")

def visit_continue_statement(self, statement: S.Continue) -> None:
  if not self.in_loop:
    gustav.error(statement.keyword, "Can't use 'continue' outside of a loop")
```

The resolver tracks loop nesting through a boolean flag, providing early error detection for misplaced control flow statements.

3.4 Built-in Function Integration

Gustav provides a clean interface for integrating built-in functions with the interpreter:

```
@t.runtime_checkable
class GusCallable(t.Protocol):
  def arity(self) -> int: ...
  def call(self, interpreter: CanExecuteBlock, arguments: list[t.Any]) -> t.Any: ...
```

Built-in functions implement this protocol, enabling seamless integration with user-defined functions:

```
class Clock(GusCallable):
  def arity(self) -> int:
    return 0

  def call(self, interpreter: CanExecuteBlock, arguments: list[str]) -> float:
    return time.perf_counter()

  def __repr__(self) -> str:
    return "<native fn>"
```

This design allows built-in functions to access the interpreter context while maintaining type safety and consistent calling conventions.

4. Performance Analysis and Results

4.1 Execution Characteristics

Performance analysis reveals typical tree-walk interpreter characteristics, with execution time scaling linearly with program complexity. Benchmark testing using recursive algorithms demonstrates approximately 15 times slower execution compared to native CPython implementations, consistent with interpreted language performance expectations[12].

The resolver's variable resolution optimization provides measurable performance improvements for variable-heavy programs by eliminating runtime scope traversal. Programs with deep nesting benefit most from this optimization.

4.2 Memory Usage and Optimization

The interpreter maintains reasonable memory usage through careful AST node design and environment management. Using CPython dataclasses with slots=True reduces per-instance memory overhead, while the garbage collection of unused environments prevents memory leaks during recursive execution.

4.3 Error Handling and Debugging Support

Gustav provides comprehensive error reporting across all compilation phases:

- **Lexical errors**: Unterminated strings, unexpected characters, invalid number formats
- **Syntax errors**: Missing tokens, malformed expressions, invalid statement structures
- **Semantic errors**: Undefined variables, invalid control flow, scope violations
- **Runtime errors**: Type mismatches, division by zero, undefined method calls

Each error category provides specific location information and suggested fixes, enhancing the development experience.

5. Learning Outcomes

5.1 Implementation Challenges and Solutions

The most instructive challenge of the project involved AST node identity management during variable resolution. This issue forced a closer look at the behavior of hash tables and the semantics of key identity, particularly the difference between structural and referential equality. It also highlighted subtle aspects of CPython's dataclass implementation, where autogenerated methods can inadvertently affect equality semantics. Debugging this challenge required carefully designed techniques to expose correctness issues that were otherwise easy to overlook, ultimately leading to a deeper understanding of low-level mechanics within the implementation.

5.2 Language Design Trade-offs

Throughout the implementation of different language features, several fundamental design trade-offs became apparent. Choosing dynamic typing over static typing simplified the initial implementation effort, but it shifted complexity into runtime, where additional checking was required to ensure correctness. Similarly, opting for tree-walk evaluation made the execution model clearer and easier to extend, though it inevitably came at the cost of runtime performance compared to a bytecode-based approach. Another key trade-off appeared between parse-time and runtime transformations: while some syntactic sugar such as pipe operators benefited from being resolved at parse-time for clarity and efficiency, other transformations were better deferred until runtime to preserve flexibility.

5.3 Software Engineering Practices

Beyond technical challenges, the project underscored the importance of sound software engineering practices. Type safety was reinforced through the use of comprehensive type annotations, which not only documented intent but also helped catch design errors at an early stage. Maintaining a high level of test coverage - 100% with more than one hundred test scenarios - provided confidence in correctness and robustness. A modular design approach ensured that components interacted through clear, well-defined interfaces, which in turn made independent development and refactoring much more manageable. Finally, extensive inline documentation proved invaluable, as it not only described how individual components worked

but also explained the reasoning behind design decisions, thus preserving the project's long-term maintainability.

6. Future Work

6.1 Bytecode Virtual Machine Implementation

The next planned development phase involves implementing a bytecode virtual machine variant to achieve better performance characteristics:

- Bytecode instruction set design
- Stack-based evaluation architecture
- Bytecode generation from AST
- Optimized virtual machine implementation
- Garbage collection
- Static type inference

7. Conclusion

To highlight the educational aspects of hands-on language implementation for computer science students, the Gustav interpreter project is one such example. Building from lexical analysis through runtime execution of a complete interpreter-level project gives the learner a great understanding of programming language concepts, along with software engineering skills.

The implementation challenges faced in development greatly aided the learning process and enriched the underlying understanding of complexity in software systems, as well as design trade-offs. This project thus sits at the crossroads of theoretical computer science and practical software construction, unlocking insights unattainable from coursework.

Future students can build upon this foundation in investigating compiler optimization techniques, alternative evaluation strategies, and experimental language features.

References

1. Nystrom, R. (2021). Crafting Interpreters. Genever Benning. Available online at: https://craftinginterpreters.com
2. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Addison-Wesley.
3. Cooper, K. D., & Torczon, L. (2011). Engineering a Compiler (2nd ed.). Morgan Kaufmann.
4. Appel, A. W. (2002). Modern Compiler Implementation in Java (2nd ed.). Cambridge University Press.
5. Watt, D. A., & Brown, D. F. (2000). Programming Language Processors in Java. Prentice Hall.
6. Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J., & Langendoen, K. (2012). Modern Compiler Design (2nd ed.). Springer.

Web of Scientists and Scholars: Journal of Multidisciplinary Research

7. Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann.
8. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
9. Sestoft, P. (2017). Programming Language Concepts (2nd ed.). Springer.
10. Krishnamurthi, S. (2012). Programming Languages: Application and Interpretation. Brown University.
11. Bracha, G., & Ungar, D. (2004). Mirrors: design principles for meta-level facilities of object-oriented programming languages. ACM SIGPLAN Notices, 39(10), 331-344.
12. Ierusalimschy, R., De Figueiredo, L. H., & Celes, W. (2007). The evolution of Lua. Proceedings of the third ACM SIGPLAN conference on History of programming languages, 2-1.
13. **Note**: all illustrations in this paper are taken from https://craftinginterpreters.com.

Appendix

Gustav's grammar specification using Wirth Syntax Notation which can also be viewed interactively on https://matthijsgroen.github.io/ebnf2railroad/try-yourself.html:

```
program = { declaration } , "EOF" ;

declaration
  = class_declaration
  | fun_declaration
  | var_declaration
  | statement
  ;
class_declaration = "class" , IDENTIFIER ,
  [ "<" , IDENTIFIER ] , "{" , { function } , "}" ;
fun_declaration = "fun" , function ;
var_declaration = "var" , IDENTIFIER , [ "=" ,
  expression ] , ";" ;
statement =
    expr_statement   | for_statement
  | if_statement       | print_statement
  | return_statement | while_statement
  | loop_statement   | block
  | break_statement  | continue_statement
  ;

expr_statement = expression , ";" ;
for_statement = "for" , "(" , ( var_declaration | expr_statement | ";" ) ,
  [ expression ] , ";" , [ expression ] , ")" ,
  statement ;
if_statement = "if" , "(" , expression , ")" ,
  statement , [ "else" , statement ] ;
print_statement = "print" , expression , ";" ;
return_statement = "return" , [ expression ] , ";" ;
while_statement = "while" , "(" , expression , ")" ,
  statement ;
```

```
loop_statement = "loop" , statement ;
block = "{" , { declaration } , "}" ;
break_statement = "break" , ";" ;
continue_statement = "continue" , ";" ;
expression = assignment ;
assignment
  = ( call , "." , IDENTIFIER , "=" , assignment )
  | logic_or
  ;
logic_or = logic_and , { "or" , logic_and } ;
logic_and = equality , { "and" , equality } ;
equality = comparison , { ( "!=" | "==" ) , comparison } ;
comparison = term , { ( ">" | ">=" | "<" | "<=" ) , term } ;
term = factor , { ( "-" | "+" | "++" | "^" ) , factor } ;
factor = unary , { ( "/" | "*" ) , unary } ;
unary = ( "!" | "-" ) , unary | call ;
call = primary , {
  ( "(" , [ arguments ] , ")"
  | "." , IDENTIFIER
  ) } , { "|>" , call } ;
primary
  = "true"              | "false"
  | "nil"               | "this"
  | NUMBER                      | STRING
  | IDENTIFIER                  | "(" , expression , ")"
  | "super" , "." , IDENTIFIER | lambda_expression
  ;
ternary = equality , [ "?" , assignment , ":" ,
  assignment ] ;
lambda_expression = ( "lambda" | "λ" ) , "(" ,
  [ parameters ] , ")" , block ;
function = IDENTIFIER , "(" , [ parameters ] , ")" , block ;
parameters = IDENTIFIER , { "," , IDENTIFIER } ;
arguments = expression , { "," , expression } ;
NUMBER = "number" ;
STRING = "string" ;
IDENTIFIER = "id" ;
```