

Bucceolang Language Documentation

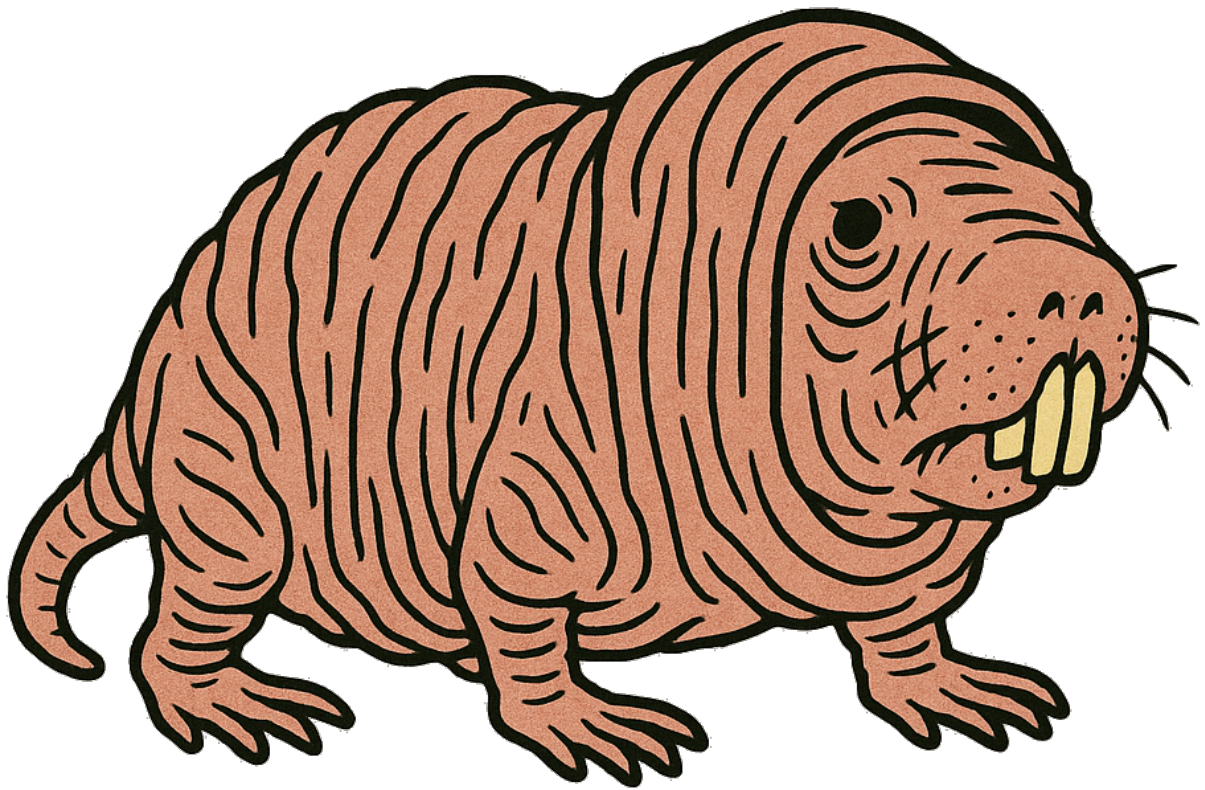


Table of contents

Bucceolang Language Documentation	I
Introduction	III
Goals	III
Language Overview	IV
Syntax	IV
Typing model	IV
Execution model	IV
Operators and Precedence	VII
Installation & Usage	VIII
Command Line Interface	VIII
Web Interface	VIII
Development Environment	VIII
Language Features	IX
Keywords	IX
Types	IX
Functions	IX
Control Flow	IX
Classes	X
Standard Library	XI
Math Functions	XII
Time Functions	XV
String Functions	XV
System Functions	XVI
Network Functions	XVII
Error Handling	XVII
Best Practices	XVIII
Advanced Examples	XIX
Architecture of the Interpreter	XXI
Module structure	XXI
Interpreter (Execution)	XXI
Error handling	XXI
Garbage collection mechanism	XXI
Appendix	XXII
Grammar specification	XXII

Introduction

Buceolang is a dynamically typed programming language with the goal of creating a secure, extensible interpreter that runs both locally and in the browser. Inspired by the concepts presented in **Crafting Interpreters**, the interpreter was implemented in Rust to leverage strong performance and efficient memory management.

The language supports core data types such as `number`, `string`, `boolean`, and `nil`, and includes lexical scoping, first-class functions, and class definitions. It also provides native functions. Thanks to WebAssembly compilation, Buceolang code can be executed directly in the browser, enabling a wide range of use cases.

This documentation provides a comprehensive overview of Buceolang, covering its language features, interpreter architecture, usage via the command line and browser, as well as technical implementation details.

Goals

The development of Buceolang was driven by the ambition to design a simple yet powerful interpreted language that is accessible for experimentation and not necessarily robust enough for real-world use cases. The current implementation establishes a solid foundation, but the project remains actively evolving.

One of the key long-term goals is to gradually extend the language with additional features such as arrays, dictionaries, and user-defined modules. Enhancing the standard library and improving error diagnostics are also on the roadmap.

Furthermore, there is strong interest in evolving the current interpreter into a more performance-oriented architecture. This could involve the implementation of a bytecode virtual machine to replace the AST-walking interpreter. Alternatively, or additionally, Buceolang could be transformed into a compiled language with its own toolchain and optimizer.

The vision for Buceolang is to maintain a steadily growing ecosystem, with continuous improvements in language expressiveness, runtime performance, and developer experience.

Language Overview

Buceolang is a modern, dynamically-typed language that combines simplicity with powerful programming constructs. This section outlines the core aspects of the language design.

Syntax

The syntax of Buceolang is designed to be clean and intuitive, taking inspiration from popular languages while maintaining its own identity. Here are the key syntax elements:

- **Statements** end with semicolons (;)
- **Blocks** are delimited by curly braces { }
- **Comments** use // for single-line and /* */ for multi-line
- **Variables** are declared using the var keyword
- **Functions** are declared using the fn keyword
- **Classes** are declared using the class keyword

Example:

```
// Single-line comment
/* Multi-line
   comment */
var name = "Alice";
fn greet(person) {
    return "Hello, " + person;
}
```

Typing model

Buceolang employs a dynamic typing system where type checking happens at runtime. This means:

- Variables can hold values of any type
- Type conversions are performed automatically when possible
- Type errors are caught during program execution
- No explicit type annotations are required

The language includes the following basic types:

- **number**: 64-bit floating-point numbers
- **string**: UTF-8 encoded text
- **boolean**: true/false values
- **nil**: represents absence of a value
- **function**: first-class function values
- **class**: user-defined types

Execution model

The execution model follows these core principles:

1. **Sequential Execution**: Code is executed top to bottom in a predictable order
2. **Lexical Scoping**: Variables are resolved in their declaring scope, enabling closures and encapsulation
3. **Garbage Collection**: Memory is automatically managed using Rust's reference counting
4. **Single-threaded**: Programs run in a single thread of execution for simplicity and safety

Code is processed through several sophisticated phases:

1. Scanning (Tokenization)

The scanner (lexer) breaks the source code into tokens through the following steps:

- Reading source code character by character
- Identifying token boundaries (whitespace, operators, etc.)
- Classifying tokens (keywords, identifiers, literals, etc.)
- Tracking line and column numbers for error reporting
- Handling comments and whitespace appropriately

Example:

```
var x = 42;
```

Becomes tokens: [VAR, IDENTIFIER("x"), EQUAL, NUMBER(42), SEMICOLON]

2. Parsing (AST Generation)

The parser converts the token stream into an Abstract Syntax Tree (AST):

- Implements recursive descent parsing
- Enforces operator precedence and associativity
- Validates syntactic structure
- Creates nodes for expressions, statements, and declarations
- Provides detailed syntax error reporting

Example AST for `x = 2 + 3`:

```
AssignmentExpr
├─ Variable(x)
└─ BinaryExpr
   ├── Number(2)
   ├── Operator(+)
   └─ Number(3)
```

3. Static Analysis (Variable Resolution)

The resolver performs semantic analysis before execution:

- Resolves variable references to their declarations
- Validates scope rules and variable usage
- Detects use of variables in their own initializers
- Ensures return only appears in functions
- Ensures break and continue only appears in loops
- Verifies this and super is only used in methods
- Builds scope chains for closures
- Reports static semantic errors

4. Interpretation (Execution)

The interpreter traverses the AST and executes the program:

a) Environment Setup:

- Creates global scope
- Defines native functions
- Initializes runtime system

b) Statement Execution:

- Evaluates expressions
- Manages variable environments
- Handles control flow
- Performs dynamic type checking
- Reports runtime errors

c) Memory Management:

- Tracks object references
- Automatically frees unused memory
- Manages method closures
- Handles cyclic references

Example execution flow:

```
// Source code
```

```
var x = 1;  
{  
    var y = x + 1;  
    print y;  
}
```

```
// Execution steps:
```

1. Create variable 'x' in global scope
2. Store value 1 in 'x'
3. Create new block scope
4. Create variable 'y' in block scope
5. Read value of 'x' from global scope (1)
6. Add 1 to value of 'x'
7. Store result (2) in 'y'
8. Output value of 'y'
9. Discard block scope

Each phase maintains detailed error information including:

- Line and column numbers
- Error context and descriptions
- Recovery strategies where applicable
- Suggestions for fixes when possible

Operators and Precedence

Buceolang supports a variety of operators with well-defined precedence rules:

Arithmetic Operators

- `+`: Addition (also used for string concatenation)
- `-`: Subtraction or unary negation
- `*`: Multiplication
- `/`: Division

Comparison Operators

- `==`: Equality
- `!=`: Inequality
- `<`: Less than
- `<=`: Less than or equal
- `>`: Greater than
- `>=`: Greater than or equal

Logical Operators

- `&&`: Logical AND
- `||`: Logical OR
- `!`: Logical NOT

Operator Precedence

From highest to lowest:

1. Unary (`!`, `-`)
2. Multiplication and Division (`*`, `/`)
3. Addition and Subtraction (`+`, `-`)
4. Comparison (`<`, `<=`, `>`, `>=`)
5. Equality (`==`, `!=`)
6. Logical AND (`&&`)
7. Logical OR (`||`)
8. Assignment (`=`)

Installation & Usage

Bucceolang can be used either through its command-line interface or via the web browser interface.

Command Line Interface

To install and run Bucceolang locally:

1. Ensure you have Rust installed (stable channel)
2. Clone the repository:

```
git clone https://github.com/m4ster-slave/bucceolang
cd bucceolang
```

3. Build the project:

```
cargo build --release
```

4. Run the REPL:

```
cargo run
```

5. Execute a script:

```
cargo run -- path/to/script.bl
```

Web Interface

The web interface provides a convenient way to write and execute Bucceolang code in your browser:

1. Visit the official Bucceolang playground
2. Use the code editor to write your program
3. Click “Run” to execute the code
4. View output in the terminal window

The web interface includes:

- Example programs
- Real-time execution
- Error reporting

Development Environment

For the best development experience:

- Use neovim

Language Features

Keywords

this

The `this` keyword is used within class methods to refer to the current instance of the class. It allows access to the instance's variables and methods.

super

The `super` keyword is used in class inheritance to call methods of the parent class. It allows a subclass to access and extend the functionality of its superclass.

Types

Buceolang provides a rich set of built-in types to support various programming needs:

Basic Types

- **Number:** 64-bit floating-point numbers used for all numeric operations
- **String:** UTF-8 encoded text strings with support for concatenation
- **Boolean:** `true` and `false` values for logical operations
- **Nil:** represents the absence of a value, similar to `null` in other languages

Complex Types

- **Functions:** First-class values that can be assigned to variables and passed as arguments
- **Classes:** User-defined types that can contain methods and instance variables

Functions

Functions in Buceolang are first-class citizens, meaning they can be:

- Assigned to variables
- Passed as arguments to other functions
- Returned from functions
- Stored in data structures

Function Declaration

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

Features

- **Lexical Scoping:** Functions create their own scope and can access variables from their enclosing scope
- **Closures:** Functions can capture and remember their enclosing scope
- **Methods:** Functions can be defined as members of classes
- **Variable Arguments:** Functions can accept any number of arguments

Control Flow

Buceolang supports standard control flow constructs:

Conditionals

```
if (condition) {  
    // code  
} else if (another_condition) {  
    // code  
} else {
```

```
    // code
}
```

Loops

- **While Loop:**

```
while (condition) {
    // code
}
```

- **For Loop:**

```
for (var i = 0; i < 10; i = i + 1) {
    // code
}
```

- **Break and Continue:** Loop control statements are supported

```
while (true) {
    if (condition) break;
    if (skip_condition) continue;
}
```

Classes

Classes in Buccelolang provide a way to create custom data types with methods and instance variables.

Class Declaration

```
class Person {
    fn init(name) {
        this.name = name;
    }

    fn greet() {
        print "Hello, I'm " + this.name;
    }
}
```

Features

- **Constructor:** The init method serves as the constructor
- **Instance Methods:** Methods can access instance variables through this
- **Superclass:** Superclass Methods can be accessed with the super keyword
- **Instance Variables:** Dynamic creation of instance variables
- **Method Calls:** Methods are called using dot notation
- **Static Methods:** Methods declared with the static keyword belong to the class itself, not instances

Static Methods

Methods in a class can be declared as static using the `static` keyword before the method declaration. Static methods belong to the class itself rather than instances of the class. They can be called on the class directly without creating an instance.

Example:

```
class MathUtils {
    static fn square(x) {
        return x * x;
    }
}

print MathUtils.square(5); // Prints: 25
```

Usage Example

```
var person = Person("Alice");
person.greet(); // Prints: Hello, I'm Alice
```

Standard Library

Buceolang provides a set of built-in functions that form its standard library:

Input/Output

- **print**
 - **Be careful! print is a statement not a function, look at the example**
 - Purpose: Display values to the console
 - Arguments: One value of any type
 - Returns: nil
 - Example: `print "Hello, world!"`;
- **input**
 - Purpose: Read a line of text from standard input
 - Arguments: Prompt as a String
 - Returns: String (the input line with whitespace trimmed)
 - Example: `var name = IO.input("Input: ")`;
- **read_file**
 - Purpose: Read the contents of a file.
 - Arguments: One string (file path).
 - Returns: String (file contents).
 - Example: `print IO.read_file("example.txt")`;
- **write_file**
 - Purpose: Write data to a file.
 - Arguments: Two strings (file path, data).
 - Returns: None.
 - Example: `IO.write_file("example.txt", "Hello, World!")`;
- **append_file**
 - Purpose: Append data to a file.
 - Arguments: Two strings (file path, data).
 - Returns: None.
 - Example: `IO.append_file("example.txt", "More data.")`;
- **exists**

- Purpose: Check if a file or directory exists.
- Arguments: One string (file path).
- Returns: Boolean.
- Example: `print IO.exists("example.txt");`
- **is_file**
 - Purpose: Check if a path is a file.
 - Arguments: One string (file path).
 - Returns: Boolean.
 - Example: `print IO.is_file("example.txt");`
- **is_dir**
 - Purpose: Check if a path is a directory.
 - Arguments: One string (directory path).
 - Returns: Boolean.
 - Example: `print IO.is_dir("example_dir");`
- **mkdir**
 - Purpose: Create a new directory.
 - Arguments: One string (directory path).
 - Returns: None.
 - Example: `IO.mkdir("new_dir");`
- **list_dir**
 - Purpose: List the contents of a directory.
 - Arguments: One string (directory path).
 - Returns: String (comma-separated list of contents).
 - Example: `print IO.list_dir("example_dir");`
- **remove_file**
 - Purpose: Delete a file.
 - Arguments: One string (file path).
 - Returns: None.
 - Example: `IO.remove_file("example.txt");`

Math Functions

- **abs**
 - Purpose: Calculate the absolute value of a number.
 - Arguments: One number.
 - Returns: Number (absolute value).
 - Example: `print Math.abs(-5);`
- **sqrt**
 - Purpose: Calculate the square root of a number.
 - Arguments: One non-negative number.
 - Returns: Number (square root).
 - Example: `print Math.sqrt(16);`
- **pow**
 - Purpose: Raise a number to a power.
 - Arguments: Two numbers (base, exponent).
 - Returns: Number (result).
 - Example: `print Math.pow(2, 3);`

- **exp**
 - Purpose: Calculate e raised to the power of a number.
 - Arguments: One number.
 - Returns: Number (e^x).
 - Example: `print Math.exp(1);`
- **log**
 - Purpose: Calculate the natural logarithm (base e) of a number.
 - Arguments: One positive number.
 - Returns: Number (natural logarithm).
 - Example: `print Math.log(2.71828);`
- **log10**
 - Purpose: Calculate the base-10 logarithm of a number.
 - Arguments: One positive number.
 - Returns: Number (base-10 logarithm).
 - Example: `print Math.log10(1000);`
- **sin**
 - Purpose: Calculate the sine of a number (in radians).
 - Arguments: One number.
 - Returns: Number (sine value).
 - Example: `print Math.sin(3.14159);`
- **cos**
 - Purpose: Calculate the cosine of a number (in radians).
 - Arguments: One number.
 - Returns: Number (cosine value).
 - Example: `print Math.cos(3.14159);`
- **tan**
 - Purpose: Calculate the tangent of a number (in radians).
 - Arguments: One number.
 - Returns: Number (tangent value).
 - Example: `print Math.tan(3.14159);`
- **asin**
 - Purpose: Calculate the arcsine of a number.
 - Arguments: One number.
 - Returns: Number (arcsine value).
 - Example: `print Math.asin(0.5);`
- **acos**
 - Purpose: Calculate the arccosine of a number.
 - Arguments: One number.
 - Returns: Number (arccosine value).
 - Example: `print Math.acos(0.5);`
- **atan**
 - Purpose: Calculate the arctangent of a number.
 - Arguments: One number.
 - Returns: Number (arctangent value).
 - Example: `print Math.atan(1);`

- **atan2**
 - Purpose: Calculate the arctangent of the quotient of two numbers.
 - Arguments: Two numbers (y, x).
 - Returns: Number (arctangent value).
 - Example: `print Math.atan2(1, 1);`
- **floor**
 - Purpose: Round a number down to the nearest integer.
 - Arguments: One number.
 - Returns: Number (rounded down).
 - Example: `print Math.floor(3.7);`
- **ceil**
 - Purpose: Round a number up to the nearest integer.
 - Arguments: One number.
 - Returns: Number (rounded up).
 - Example: `print Math.ceil(3.1);`
- **round**
 - Purpose: Round a number to the nearest integer.
 - Arguments: One number.
 - Returns: Number (rounded).
 - Example: `print Math.round(3.5);`
- **trunc**
 - Purpose: Truncate a number to its integer part.
 - Arguments: One number.
 - Returns: Number (truncated).
 - Example: `print Math.trunc(3.9);`
- **degrees**
 - Purpose: Convert radians to degrees.
 - Arguments: One number.
 - Returns: Number (degrees).
 - Example: `print Math.degrees(3.14159);`
- **radians**
 - Purpose: Convert degrees to radians.
 - Arguments: One number.
 - Returns: Number (radians).
 - Example: `print Math.radians(180);`
- **min**
 - Purpose: Find the minimum of two numbers.
 - Arguments: Two numbers.
 - Returns: Number (minimum value).
 - Example: `print Math.min(3, 5);`
- **max**
 - Purpose: Find the maximum of two numbers.
 - Arguments: Two numbers.
 - Returns: Number (maximum value).
 - Example: `print Math.max(3, 5);`

- **clamp**
 - Purpose: Clamp a number within a range.
 - Arguments: Three numbers (value, min, max).
 - Returns: Number (clamped value).
 - Example: `print Math.clamp(5, 1, 10);`
- **random**
 - Purpose: Generate a random number in range [0, max).
 - Arguments: One positive number (max).
 - Returns: Number (random value).
 - Example: `print Math.random(10);`

Time Functions

- **time**
 - Purpose: Get the current Unix timestamp.
 - Arguments: None.
 - Returns: Number (seconds since Unix epoch).
 - Example: `print Time.time();`
- **sleep**
 - Purpose: Pause execution for a specified number of seconds.
 - Arguments: One non-negative number (seconds).
 - Returns: None.
 - Example: `Time.sleep(2);`

String Functions

- **len**
 - Purpose: Get the length of a string.
 - Arguments: One string.
 - Returns: Number (length of the string).
 - Example: `print String.len("hello");`
- **split**
 - Purpose: Split a string into parts based on a separator.
 - Arguments: Two strings (string to split, separator).
 - Returns: String (comma-separated parts).
 - Example: `print String.split("a,b,c", ",");`
- **join**
 - Purpose: Join parts of a string with a separator.
 - Arguments: Two strings (separator, comma-separated parts).
 - Returns: String (joined string).
 - Example: `print String.join("-", "a,b,c");`
- **replace**
 - Purpose: Replace occurrences of a substring with another substring.
 - Arguments: Three strings (original string, substring to replace, replacement).
 - Returns: String (modified string).
 - Example: `print String.replace("hello world", "world", "rust");`
- **lower**
 - Purpose: Convert a string to lowercase.
 - Arguments: One string.

- Returns: String (lowercase version).
- Example: `print String.lower("HeLLo");`
- **upper**
 - Purpose: Convert a string to uppercase.
 - Arguments: One string.
 - Returns: String (uppercase version).
 - Example: `print String.upper("HeLLo");`
- **strip**
 - Purpose: Remove leading and trailing whitespace from a string.
 - Arguments: One string.
 - Returns: String (trimmed string).
 - Example: `print String.strip(" hi ");`
- **startswith**
 - Purpose: Check if a string starts with a prefix.
 - Arguments: Two strings (string, prefix).
 - Returns: Boolean.
 - Example: `print String.startswith("foobar", "foo");`
- **endswith**
 - Purpose: Check if a string ends with a suffix.
 - Arguments: Two strings (string, suffix).
 - Returns: Boolean.
 - Example: `print String.endswith("foobar", "bar");`
- **find**
 - Purpose: Find the index of a substring in a string.
 - Arguments: Two strings (string, substring).
 - Returns: Number (index) or nil if not found.
 - Example: `print String.find("hello world", "world");`
- **contains**
 - Purpose: Check if a string contains a substring.
 - Arguments: Two strings (string, substring).
 - Returns: Boolean.
 - Example: `print String.contains("hello world", "world");`

System Functions

- **exit**
 - Purpose: Exit the program with a status code.
 - Arguments: One number (status code).
 - Returns: None (terminates the program).
 - Example: `System.exit(0);`
- **env**
 - Purpose: Get the value of an environment variable.
 - Arguments: One string (variable name).
 - Returns: String (value) or nil if not found.
 - Example: `print System.env("PATH");`
- **args**
 - Purpose: Get the command-line arguments.

- Arguments: None.
- Returns: String (comma-separated arguments).
- Example: `print System.args();`
- **exec**
 - Purpose: Execute a shell command.
 - Arguments: One string (command).
 - Returns: String (command output).
 - Example: `print System.exec("echo hi");`
- **platform**
 - Purpose: Get the current operating system.
 - Arguments: None.
 - Returns: String (OS name).
 - Example: `print System.platform();`

Network Functions

- **http_get**
 - Purpose: Perform an HTTP GET request.
 - Arguments: One string (URL).
 - Returns: String (response body).
 - Example: `print Network.http_get("https://example.com");`
- **http_post**
 - Purpose: Perform an HTTP POST request.
 - Arguments: Two strings (URL, body).
 - Returns: String (response body).
 - Example: `print Network.http_post("https://example.com", "key=value");`
- **download_file**
 - Purpose: Download a file from a URL.
 - Arguments: Two strings (URL, file path).
 - Returns: None.
 - Example: `Network.download_file("https://example.com/file.txt", "local_file.txt");`
- **ping**
 - Purpose: Ping a host to check connectivity.
 - Arguments: One string (host).
 - Returns: Boolean (success or failure).
 - Example: `print Network.ping("8.8.8.8");`

Error Handling

Bucceolang provides comprehensive error detection and reporting:

Compile-time Errors

1. Syntax Errors

- Invalid token sequences
- Mismatched parentheses/braces
- Invalid variable names
- Example: `var 123abc = 5; // Invalid variable name`

2. Resolution Errors

- Variable used in its own initializer

- Return statement outside function
- Break/continue outside loop
- Example: `var a = a; // Variable used in own initializer`

Runtime Errors

1. Type Errors

- Invalid operand types
- Example: `"hello" - 5 // Can't subtract number from string`

2. Reference Errors

- Undefined variables
- Undefined properties
- Example: `print undefined_var; // Reference to undefined variable`

3. Value Errors

- Division by zero
- Invalid argument types to native functions
- Example: `sqrt(-1) // Negative argument to sqrt`

Error Recovery

The interpreter provides error recovery mechanisms to:

- Continue parsing after syntax errors
- Maintain environment consistency after runtime errors
- Provide meaningful error messages with line numbers
- Example error message: "Line 5: Undefined variable 'counter'"

Best Practices

When writing Buccelolang code, follow these guidelines:

1. Variable Naming

- Use descriptive names
- Start with lowercase letter
- Use snake case for multi-word names

```
var first_name = "John";
var max_attempts = 3;
```

2. Function Organization

- Keep functions small and focused
- Use meaningful parameter names

```
fn calculate_area(width, height) {
    return width * height;
}
```

3. Class Design

- Initialize all instance variables in init
- Use clear method names

```
class BankAccount {
    fn init(balance) {
        this.balance = balance;
    }

    fn deposit(amount) {
        this.balance = this.balance + amount;
    }
}
```

```
    }
}
```

4. Error Handling

- Check function arguments
- Handle edge cases

```
fn divide(a, b) {
    if (b == 0) {
        print "Error: Division by zero";
        return nil;
    }
    return a / b;
}
```

Advanced Examples

Closure Example

```
fn make_counter() {
    var count = 0;
    fn counter() {
        count = count + 1;
        return count;
    }
    return increment;
}

var counter = make_counter();
print counter(); // 1
print counter(); // 2
```

Class Inheritance

```
class Shape {
    fn get_area() {
        return 0;
    }
}

class Rectangle < Shape {
    fn init(width, height) {
        this.width = width;
        this.height = height;
    }

    fn get_area() {
        return this.width * this.height;
    }
}
```

Complex Data Structure

```
class Node {
    fn init(value) {
        this.value = value;
        this.next = nil;
    }
}

class LinkedList {
    fn init() {
        this.head = nil;
    }

    fn add(value) {
        var node = Node(value);
        if (this.head == nil) {
            this.head = node;
        } else {
            var current = this.head;
            while (current.next != nil) {
                current = current.next;
            }
            current.next = node;
        }
    }

    fn print_list() {
        var current = this.head;
        while (current != nil) {
            print current.value;
            current = current.next;
        }
    }
}

var list = LinkedList();
list.add(1);
list.add(2);
list.add(3);
list.print_list();
```

Architecture of the Interpreter

Module structure

Scanner (Tokenization)

The scanner breaks down source code into tokens. It handles:

- Keywords and identifiers
- Numeric and string literals
- Operators and punctuation
- Comments and whitespace

Parser (AST generation)

The parser constructs an Abstract Syntax Tree (AST) from tokens:

- Expression parsing
- Statement parsing
- Error recovery and reporting
- Precedence and associativity handling

Resolver (Scope resolution)

The resolver performs static analysis:

- Variable declaration checking
- Scope analysis
- Class and function validation
- This binding validation

Interpreter (Execution)

The interpreter evaluates the AST:

- Expression evaluation
- Statement execution
- Environment management
- Runtime error handling

Error handling

The interpreter provides comprehensive error handling:

- **Syntax Errors:** Detected during parsing
- **Runtime Errors:** Caught during execution
- **Resolution Errors:** Found during static analysis
- **Type Errors:** Detected during runtime

Garbage collection mechanism

Memory management in Buccelolang is automatic:

- Reference counting using Rust's Rc
- Automatic cleanup of unreferenced values
- Proper handling of circular references
- Deterministic cleanup of resources

Appendix

Grammar specification

The following EBNF-style grammar defines the syntax of Buccelang:

```
program      → declaration* EOF ;

declaration  → classDecl
             | funDecl
             | varDecl
             | statement ;

classDecl    → "class" IDENTIFIER "{" function* "}" ;
funDecl      → "fn" function ;
varDecl      → "var" IDENTIFIER ( "=" expression )? ";" ;

statement    → exprStmt
             | forStmt
             | ifStmt
             | printStmt
             | returnStmt
             | whileStmt
             | block ;

expression   → assignment ;
assignment   → ( call "." )? IDENTIFIER "=" assignment
             | logic_or ;

primary      → "true" | "false" | "nil" | "this"
             | NUMBER | STRING | IDENTIFIER | "(" expression ")"
             | "super" "." IDENTIFIER ;
```