Interpreter Implementation Checklist

Phase 1: Core Data Structures

1.1 Value Representation

- [ ] Define Value enum with all variants (Integer, UnsignedInt, Float, Boolean, String, Array, Struct, Pointer, Enum, Null)
- [ ] Implement PointerValue struct with target_id, is_const, is_null
- [ ] Implement StructInstance struct with struct_id, fields HashMap, heap_id
- [ ] Implement EnumValue struct with variant and optional value
- [ ] Create Display/Debug traits for Value type

1.2 Type System

- [ ] Define Type enum (I32, U32, F32, Bool, String, Array, Struct, Enum, Pointer)
- [ ] Implement type equality checking
- [ ] Create type-to-string conversion utilities
- [ ] Add type inference for literals

1.3 AST Nodes

- [ ] Define Expression enum (Literal, Variable, BinaryOp, UnaryOp, Call, Index, FieldAccess, AddressOf, Dereference)
- [ ] Define Statement enum (VarDecl, Assignment, If, For, While, Return, Break, Continue, ExprStmt, Block)
- [ ] Define Declaration enum (Function, Struct, Enum)
- [ ] Create Program structure to hold declarations and statements

1.4 Memory Management Structures

- [ ] Define HeapObjectId as u64 wrapper
- [ ] Implement HeapObject struct with all fields
- [ ] Create AllocationStrategy enum (RefCounted, Traced)
- [ ] Define BorrowError enum with all variants
- [ ] Define GCError enum

Phase 2: Lexer & Tokenizer

2.1 Token Definition

- [ ] Define Token enum with all token types
- [ ] Create TokenType enum (Identifier, Keyword, IntLiteral, FloatLiteral, StringLiteral, BoolLiteral, Operator, Separator, Comment)
- [ ] Implement Token struct with type, lexeme, literal, line, column

2.2 Lexer Implementation

- [ ] Create Lexer struct with source code and position tracking
- [ ] Implement character scanning methods
- [ ] Implement identifier and keyword recognition
- [ ] Implement integer literal scanning
- [ ] Implement float literal scanning
- [ ] Implement string literal scanning with escape sequences
- [ ] Implement operator recognition
- [ ] Implement separator recognition
- [ ] Implement comment skipping
- [ ] Implement whitespace handling
- [ ] Create tokenize() method to return Vec<Token>
- [ ] Add error reporting for invalid tokens

Phase 3: Parser Basics

3.1 Parser Infrastructure

- [ ] Create Parser struct with tokens and current position
- [ ] Implement peek() for lookahead
- [ ] Implement advance() for consuming tokens
- [ ] Implement check() for token type checking
- [ ] Implement match_token() for conditional consumption
- [ ] Implement error() method with line/column info

3.2 Expression Parsing

- [ ] Implement parse_expression() entry point
- [ ] Implement parse_assignment() with = operator
- [ ] Implement parse_logical_or()
- [ ] Implement parse_logical_and()
- [ ] Implement parse_equality() with ==, !=
- [ ] Implement parse_comparison() with <, >, <=, >=
- [ ] Implement parse_addition() with +, -
- [ ] Implement parse_multiplication() with *, /, %
- [ ] Implement parse_exponentiation() with ^, **
- [ ] Implement parse_unary() with -, !, ++, --
- [ ] Implement parse_postfix() with ++, --, array indexing, field access, method calls
- [ ] Implement parse_primary() with literals, variables, parentheses, array literals
- [ ] Implement parse_address_of() with & operator
- [ ] Implement parse_dereference() with * operator

3.3 Statement Parsing

- [ ] Implement parse_statement() dispatcher
- [ ] Implement parse_var_declaration() for let/const
- [ ] Implement parse_assignment_statement()
- [ ] Implement parse_if_statement()
- [ ] Implement parse_for_loop()
- [ ] Implement parse_while_loop()
- [ ] Implement parse_return_statement()
- [ ] Implement parse_break_statement()
- [ ] Implement parse_continue_statement()
- [ ] Implement parse_block()
- [ ] Implement parse_expression_statement()

3.4 Declaration Parsing

- [ ] Implement parse_function_declaration()
- [ ] Implement parse_parameter_list()
- [ ] Implement parse_return_type()
- [ ] Implement parse_struct_declaration()
- [ ] Implement parse_struct_body()
- [ ] Implement parse_struct_field()
- [ ] Implement parse_modify_methods()
- [ ] Implement parse_static_methods()
- [ ] Implement parse_method_declaration()
- [ ] Implement parse_enum_declaration()
- [ ] Implement parse_enum_body()
- [ ] Implement parse_enum_member()

Phase 4: Borrow Checker

4.1 Borrow Checker Infrastructure

- [ ] Create BorrowChecker struct
- [ ] Implement Binding struct storage
- [ ] Implement BorrowInfo struct tracking
- [ ] Implement Borrow struct with timestamps
- [ ] Create Scope struct for scope management
- [ ] Implement bindings HashMap

4.2 Variable Binding Tracking

- [ ] Implement register_binding() for new variables
- [ ] Implement get_binding() for variable lookup
- [ ] Implement is_mutable_binding() check
- [ ] Implement is_const_binding() check
- [ ] Implement binding_type() retrieval
- [ ] Track scope levels for each binding

4.3 Borrow Rules Enforcement

- [ ] Implement check_borrow() with all three rules
- [ ] Implement cannot_mutate_const rule
- [ ] Implement cannot_mutable_with_immutable_borrows rule
- [ ] Implement cannot_immutable_with_mutable_borrow rule
- [ ] Create register_borrow() for tracking new borrows
- [ ] Implement release_borrow() for scope exit
- [ ] Track borrow creation and release timestamps

4.4 Borrow Validation

- [ ] Implement validate_pointer_usage()
- [ ] Implement check_dangling_pointer()
- [ ] Implement validate_use_after_free()
- [ ] Implement check_const_pointer_mutation()
- [ ] Validate pointer escaping scope

Phase 5: Garbage Collector

5.1 GC Infrastructure

- [ ] Create GarbageCollector struct
- [ ] Implement heap HashMap storage
- [ ] Initialize white_set, gray_set, black_set
- [ ] Implement gc_roots vector
- [ ] Implement HeapObjectId generator with next_id
- [ ] Create HeapObject storage

5.2 Allocation Strategy

- [ ] Implement allocate() with AllocationStrategy parameter
- [ ] Implement RefCounted allocation path
- [ ] Implement Traced allocation path
- [ ] Track tracked_by_rc flag on objects
- [ ] Implement should_collect() threshold logic
- [ ] Add automatic collection triggering

5.3 Reference Counting (Fast Path)

- [ ] Implement incr_ref() for reference count increment
- [ ] Implement decr_ref() for reference count decrement
- [ ] Implement automatic deallocation when refcount hits zero
- [ ] Add RefCell borrowing for RC objects

5.4 Mark-Sweep GC (Slow Path)

- [ ] Implement mark_reachable() phase
- [ ] Implement gray_set processing loop
- [ ] Implement scan_object() for object examination
- [ ] Implement mark_references() for reference discovery
- [ ] Add Array reference marking
- [ ] Add Struct field reference marking
- [ ] Add Pointer target reference marking
- [ ] Implement sweep() phase
- [ ] Implement collect() full cycle
- [ ] Add collection triggering based on traced object count

5.5 GC Root Management

- [ ] Implement register_gc_root()
- [ ] Implement unregister_gc_root()
- [ ] Track all reachable roots

Phase 6: Interpreter Core

6.1 Interpreter Structure

- [ ] Create Interpreter struct
- [ ] Integrate BorrowChecker
- [ ] Integrate GarbageCollector
- [ ] Create ExecutionStack
- [ ] Create ScopeManager
- [ ] Implement StackFrame structure

6.2 Execution Stack Management

- [ ] Implement push_frame()
- [ ] Implement pop_frame()
- [ ] Implement get_local_variable()
- [ ] Implement set_local_variable()
- [ ] Track stack frames for each scope

6.3 Scope Management

- [ ] Implement enter_scope()
- [ ] Implement exit_scope()
- [ ] Implement cleanup on scope exit
- [ ] Release borrows on scope exit
- [ ] Handle variable shadowing
- [ ] Track scope levels

Phase 7: Primitive Operations

7.1 Arithmetic Operations

- [ ] Implement addition (+) for all numeric types
- [ ] Implement subtraction (-) for all numeric types
- [ ] Implement multiplication (*) for all numeric types
- [ ] Implement division (/) for all numeric types
- [ ] Implement modulo (%) operation
- [ ] Implement power (**) operation
- [ ] Implement exponentiation (^) operation
- [ ] Handle type coercion for mixed operations
- [ ] Implement division by zero error

7.2 Comparison Operations

- [ ] Implement equality (==) for all types
- [ ] Implement inequality (!=) for all types
- [ ] Implement less than (<) for numeric types
- [ ] Implement greater than (>) for numeric types
- [ ] Implement less than or equal (<=) for numeric types
- [ ] Implement greater than or equal (>=) for numeric types

7.3 Logical Operations

- [ ] Implement logical AND (&&)
- [ ] Implement logical OR (||)
- [ ] Implement logical NOT (!)
- [ ] Implement short-circuit evaluation

7.4 Increment/Decrement

- [ ] Implement pre-increment (++var)
- [ ] Implement post-increment (var++)
- [ ] Implement pre-decrement (--var)
- [ ] Implement post-decrement (var--)
- [ ] Validate mutable binding requirement

Phase 8: Variable Declaration & Assignment

8.1 Variable Declaration

- [ ] Implement execute_var_declaration()
- [ ] Handle let declarations
- [ ] Handle const declarations
- [ ] Support type inference with :=
- [ ] Support explicit typing with =
- [ ] Handle zero initialization
- [ ] Store in current frame
- [ ] Register with borrow checker
- [ ] Allocate on heap for complex types

8.2 Variable Assignment

- [ ] Implement execute_assignment()
- [ ] Validate const protection
- [ ] Check borrow safety
- [ ] Handle RC increment/decrement
- [ ] Support compound assignment (+=, -=, *=, /=, %=)
- [ ] Update local variables

8.3 Variable Resolution

- [ ] Implement get_variable()
- [ ] Search through stack frames
- [ ] Handle variable shadowing
- [ ] Return undefined error for missing variables

Phase 9: Pointer Operations

9.1 Address-of Operator

- [ ] Implement address_of() for variables
- [ ] Validate cannot take address of primitives
- [ ] Create PointerValue with correct mutability
- [ ] Allocate pointer targets on heap
- [ ] Set is_const based on binding

9.2 Dereference Operator

- [ ] Implement dereference() for pointer values
- [ ] Check for null pointer
- [ ] Check for dangling pointer
- [ ] Return dereferenced value
- [ ] Validate pointer validity

9.3 Pointer Assignment

- [ ] Implement assign_through_pointer()
- [ ] Validate null pointer
- [ ] Validate dangling pointer
- [ ] Enforce const pointer protection
- [ ] Update heap object value

9.4 Pointer Arithmetic

- [ ] Implement pointer offset calculation
- [ ] Validate bounds checking
- [ ] Handle null pointer state

Phase 10: Arrays

10.1 Array Literals

- [ ] Implement array literal parsing
- [ ] Create arrays with element list
- [ ] Allocate on heap with RC strategy
- [ ] Return pointer to array

10.2 Array Indexing

- [ ] Implement array_index() read operation
- [ ] Validate array type
- [ ] Validate index bounds
- [ ] Return element value
- [ ] Support negative index errors

10.3 Array Assignment

- [ ] Implement array_set() write operation
- [ ] Validate mutable binding
- [ ] Validate index bounds
- [ ] Update element in heap
- [ ] Handle RC for assigned values

10.4 Dynamic Arrays

- [ ] Support dynamic sizing ([].type)
- [ ] Implement push operation (if supported)
- [ ] Implement pop operation (if supported)
- [ ] Track array length

10.5 Fixed Arrays

- [ ] Support fixed sizing ([N].type)
- [ ] Enforce size at allocation
- [ ] Validate index within bounds

Phase 11: Control Flow

11.1 If/Else Statements

- [ ] Implement execute_if_statement()
- [ ] Evaluate condition expression
- [ ] Execute then block if true
- [ ] Execute else_if blocks in order
- [ ] Execute else block if all false
- [ ] Enter/exit scope for blocks

11.2 For Loops

- [ ] Implement execute_for_loop()
- [ ] Execute initializer
- [ ] Evaluate condition
- [ ] Execute loop body
- [ ] Execute increment
- [ ] Handle break statement
- [ ] Handle continue statement
- [ ] Enter/exit loop scope

11.3 While Loops

- [ ] Implement execute_while_loop()
- [ ] Evaluate condition
- [ ] Execute loop body
- [ ] Handle break statement
- [ ] Handle continue statement
- [ ] Enter/exit loop scope

11.4 Break/Continue

- [ ] Track loop nesting level
- [ ] Implement break with immediate exit
- [ ] Implement continue with next iteration
- [ ] Validate inside loop context
- [ ] Handle scope cleanup

Phase 12: Functions

12.1 Function Definition

- [ ] Create function storage/registry
- [ ] Store function signature
- [ ] Store function body
- [ ] Store parameter information
- [ ] Store return type

12.2 Function Calls

- [ ] Implement execute_call()
- [ ] Resolve function name
- [ ] Validate argument count
- [ ] Validate argument types
- [ ] Enter new scope for call
- [ ] Bind parameters to frame
- [ ] Execute function body
- [ ] Exit scope after call

12.3 Return Statements

- [ ] Implement execute_return()
- [ ] Evaluate return expression
- [ ] Store return value
- [ ] Signal return from current scope
- [ ] Handle implicit void return

12.4 Function Parameters

- [ ] Bind parameters in new frame
- [ ] Validate parameter types
- [ ] Support mutable parameters
- [ ] Support const parameters
- [ ] Handle parameter shadowing

12.5 Function Recursion

- [ ] Support recursive calls
- [ ] Handle call stack depth

Phase 13: Structs

13.1 Struct Definition

- [ ] Create struct registry/storage
- [ ] Store struct ID
- [ ] Store struct fields with types
- [ ] Store method definitions
- [ ] Distinguish Modify vs Static methods

13.2 Struct Instantiation

- [ ] Implement create_struct_instance()
- [ ] Allocate StructInstance
- [ ] Initialize fields to zero
- [ ] Assign initial values (if provided)
- [ ] Allocate on heap
- [ ] Return pointer

13.3 Field Access

- [ ] Implement field_access() read
- [ ] Validate struct exists
- [ ] Validate field exists
- [ ] Return field value
- [ ] Support chained access

13.4 Field Assignment

- [ ] Implement field_assign() write
- [ ] Validate mutable struct
- [ ] Validate field exists
- [ ] Update field value
- [ ] Handle RC for assigned values

13.5 Instance Methods (Modify)

- [ ] Implement call_method() for instance methods
- [ ] Bind self to instance
- [ ] Bind parameters
- [ ] Enter method scope
- [ ] Execute method body
- [ ] Allow self mutation
- [ ] Exit scope and return

13.6 Static Methods

- [ ] Implement call_static_method()
- [ ] Do not bind self
- [ ] Bind parameters
- [ ] Enter method scope
- [ ] Execute method body
- [ ] Exit scope and return

Phase 14: Enums

14.1 Enum Definition

- [ ] Create enum registry/storage
- [ ] Store enum ID
- [ ] Store enum variants
- [ ] Support integer values
- [ ] Support string values
- [ ] Support zero default values

14.2 Enum Construction

- [ ] Implement create_enum_value()
- [ ] Validate variant exists
- [ ] Store variant name
- [ ] Store optional associated value
- [ ] Return EnumValue

14.3 Enum Usage

- [ ] Support enum variable assignment
- [ ] Pattern matching (if supported)
- [ ] Variant comparison

Phase 15: Type Checking

15.1 Type Inference

- [ ] Implement type_of() for all expressions
- [ ] Infer from literals
- [ ] Infer from operations
- [ ] Infer from function returns
- [ ] Infer from array elements

15.2 Type Validation

- [ ] Implement validate_types_compatible()
- [ ] Check assignment type safety
- [ ] Check function argument types
- [ ] Check operation operand types
- [ ] Report type mismatches

15.3 Implicit Conversions

- [ ] Support automatic type coercion rules
- [ ] Numeric type promotion

Phase 16: Strings

16.1 String Literals

- [ ] Parse string literals
- [ ] Handle escape sequences (\n, \t, \, ")
- [ ] Create String value
- [ ] Store as heap object

16.2 String Operations

- [ ] Support string concatenation
- [ ] Support string comparison
- [ ] Support string length
- [ ] Support string indexing

Phase 17: Error Handling

17.1 Runtime Errors

- [ ] Implement error reporting
- [ ] Include line and column information
- [ ] Show error message
- [ ] Track error type
- [ ] Graceful error recovery (if applicable)

17.2 Type Errors

- [ ] Report type mismatches
- [ ] Report undefined variables
- [ ] Report undefined functions

17.3 Borrow Errors

- [ ] Report borrow violations
- [ ] Report use-after-free
- [ ] Report dangling pointers
- [ ] Report const violations

17.4 Runtime Errors

- [ ] Report division by zero
- [ ] Report array out of bounds
- [ ] Report null pointer dereference
- [ ] Report stack overflow

Phase 18: Integration & Testing

18.1 Full Pipeline

- [ ] Integrate lexer -> parser -> interpreter
- [ ] Execute complete programs
- [ ] Test variable declarations
- [ ] Test arithmetic operations
- [ ] Test control flow
- [ ] Test function calls

18.2 Basic Test Suite

- [ ] Test primitive types
- [ ] Test variables and assignment
- [ ] Test arithmetic operations
- [ ] Test control flow (if, for, while)
- [ ] Test function definitions and calls

18.3 Intermediate Test Suite

- [ ] Test arrays
- [ ] Test structs with fields
- [ ] Test instance methods
- [ ] Test static methods
- [ ] Test pointers and references

18.4 Advanced Test Suite

- [ ] Test circular references
- [ ] Test complex object graphs
- [ ] Test scope nesting
- [ ] Test recursion
- [ ] Test error handling

Phase 19: Memory Management Validation

19.1 RC Testing

- [ ] Verify immediate deallocation at refcount zero
- [ ] Test reference counting accuracy
- [ ] Test RC with nested objects
- [ ] Verify no memory leaks (RC path)

19.2 GC Testing

- [ ] Verify mark-sweep correctness
- [ ] Test object reachability detection
- [ ] Test cycle detection
- [ ] Test collection triggering
- [ ] Verify no memory leaks (GC path)

19.3 Hybrid Testing

- [ ] Test RC + GC interaction
- [ ] Test objects transitioning paths
- [ ] Verify allocation strategy selection

Phase 20: Optimization

20.1 Performance Analysis

- [ ] Profile interpreter execution
- [ ] Identify hot paths
- [ ] Measure memory usage

20.2 RC Optimizations

- [ ] Implement move semantics
- [ ] Batch RC operations
- [ ] Use weak references for cycles
- [ ] Stack allocate short-lived objects

20.3 GC Optimizations

- [ ] Implement generational GC
- [ ] Add write barriers
- [ ] Implement incremental collection
- [ ] Add pool allocation

20.4 General Optimizations

- [ ] Cache frequently accessed values
- [ ] Optimize type checking
- [ ] Reduce allocation overhead

Phase 21: Advanced Features (Optional)

21.1 Pattern Matching

- [ ] Implement enum pattern matching
- [ ] Support nested patterns
- [ ] Generate match errors

21.2 Error Propagation

- [ ] Implement optional/error types
- [ ] Support unwrap operations
- [ ] Support default values

21.3 Iterators/Ranges

- [ ] Implement range support
- [ ] Implement iterator protocol
- [ ] Support for-in loops

21.4 Generics (If Designed)

- [ ] Generic struct definitions
- [ ] Generic function definitions
- [ ] Type parameter constraints
- [ ] Monomorphization

Phase 22: Documentation & Refinement

22.1 Code Documentation

- [ ] Document public APIs
- [ ] Document error types
- [ ] Document memory model

22.2 Usage Examples

- [ ] Create example programs
- [ ] Test with fibonacci
- [ ] Test with data structures
- [ ] Test with recursive algorithms

22.3 Final Validation

- [ ] Comprehensive integration testing
- [ ] Performance benchmarking
- [ ] Memory leak verification
- [ ] Error handling verification