21 Apr 2025
Koh Chan Hong, A0254616A
Dexter Leng, A0273293Y

C4215 Term Project Report

Project repository	https://github.com/ravern/shiny
End-to-end test cases	https://github.com/ravern/shiny/tree/main/tests/e2e

Contents

21 Apr 2025
Koh Chan Hong, A0254616A
Dexter Leng, A0273293Y

1. Overview

This project implements an interpreter for a subset of Swift, which we call Shiny, using C++. It comprises two major components: a compiler that translates Swift source code into custom bytecode, and a virtual machine (VM) that executes this bytecode. The primary goal is to understand the internals of language implementation, from parsing to execution, with a focus on Swift's semantics.

1.1. Project Objectives

In order to gain a deeper understanding of language implementation, we aim to achieve the following objectives in this project:

- Implement a subset of Swift's language features (specified in Section 1.2).
- Design a compiler that parses Shiny source code and emits bytecode instructions.
- Build a stack-based virtual machine to execute the bytecode.
- Support runtime features such as control flow, lexical scoping and closures.
- Implement reference-counted objects for memory management similar to Swift's ARC
- Explore type checking and memory management techniques relevant to interpreter design.

1.2. Extent of Progress

We have implemented a working set of components that together form the Shiny interpreter from end-to-end. These components are:

- A recursive descent parser for a our subset of Swift grammar
- A type checker with support for local type inference
- A working bytecode compiler, from Swift AST to custom bytecode instructions
- A VM to interpret the bytecode
- Reference-counted objects wrapped in a convenient C++ API

At the point of submission, the following Swift language features have been implemented:

- Comments
- Basic data types (Bool, Int, Double)
- Local and global variables
- Control flow (if)
- Top-level functions, recursion
- Classes, object instantiation, methods
- Closures

1.3. Platform Used

Shiny is a single binary executable that contains both an interactive REPL environment and an interpreter for a Shiny file.

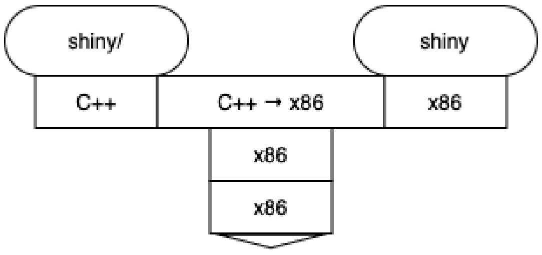

Figure 1: Diagram showing how the Shiny program is compiled

Figure 1 shows how the Shiny program is compiled using the C++ compiler available on the platform (in our case, clang++) to a single executable called `shiny`.

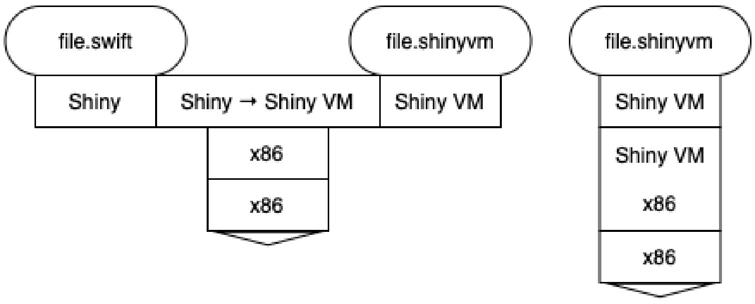

Figure 2: Diagram showing the internal components of the Shiny program

Figure 2 shows the breakdown of how the compiled `shiny` (containing x86 machine code) runs a given `file.swift` by first compiling it to bytecode (represented as the "language" shinyvm) and then running that bytecode.

1.3.1. Software Used

- Programming Language: C++20
- Build System: CMake
- Operating System: macOS
- Development Tools: clang++, g++
- External Libraries: C++ Standard Template Library, C standard library
- Version Control: git
- Collaboration: GitHub

2. Grammar

Parsing is done with a one-token look-ahead recursive descent algorithm.

2.1. Extended Backus-Naur Form

```
<program> ::= { <statement> }

<statement> ::= <declare-stmt>
              | <return-stmt>
              | <class-stmt>
              | <function-stmt>
              | <if-stmt>
              | <expression-stmt>
```

```
<declare-stmt> ::= "var" <identifier> "=" <expression>

<return-stmt> ::= "return" <expression>?

<class-stmt> ::= "class" <identifier> "{" { <class-member> } "}"

<class-member> ::= <declare-stmt> | <function-stmt>

<function-stmt> ::= "func" <identifier> "(" [ <param-list> ] ")" [ "->" <type> ] <block>

<param-list> ::= <param> { "," <param> }

<param> ::= <identifier> ":" <type>

<if-stmt> ::= "if" <expression> <block> [ "else" <else-branch> ]

<else-branch> ::= <if-stmt> | <block>

<expression-stmt> ::= <expression>

<block> ::= "{" { <statement> } "}"

<expression> ::= <assignment>

<assignment> ::= <logical-or> [ "=" <assignment> ]

<logical-or> ::= <logical-and> { "||" <logical-and> }

<logical-and> ::= <equality> { "&&" <equality> }

<equality> ::= <comparison> { ("==" | "!=") <comparison> }

<comparison> ::= <term> { ("<" | "<=" | ">" | ">=") <term> }

<term> ::= <factor> { ("+" | "-") <factor> }

<factor> ::= <unary> { ("*" | "/" | "%") <unary> }

<unary> ::= ("!" | "-") <unary> | <call>

<call> ::= <primary> { <call-suffix> }

<call-suffix> ::= "(" [ <expression> { "," <expression> } ] ")"
                | "." <identifier>

<primary> ::= <literal>
            | "self"
            | <identifier>
            | "(" <expression>? ")"

<literal> ::= <int>
            | <float>
            | "true"
            | "false"
```

```
<type> ::= "Int" | "Double" | "Bool"
         | "()"                         % Void type
         | "(" <type> ")"               % grouped type e.g. (Int)
         | <function-type>

<function-type> ::= "(" [ <type-list> ] ")" "->" <type>
<type-list> ::= <type> { "," <type> }
```

2.2. No semicolons

In Shiny, semicolons are not used to terminate statements. Instead, the parser relies on line breaks to infer statement boundaries. To support this, each token is annotated with a boolean flag `isAtStartOfLine`, which indicates whether the token is the first non-whitespace token to appear on a new line of source code.

This flag enables the parser to enforce layout-sensitive rules that cannot be captured by a traditional context-free grammar. For example, the parser ensures that top-level statements such as variable declarations, return statements, class definitions, and function definitions must begin on a new line. Similarly, the presence or absence of a new line after a return keyword determines whether it is a return Void or a return .

This behavior is context-sensitive: it depends not only on the sequence of tokens but also on their physical placement within the source text. Since Extended Backus-Naur Form (EBNF) describes only syntactic structure and not source layout, it cannot express constraints like "this token must appear at the start of a new line." These constraints are instead implemented directly in the parser using the `isAtStartOfLine` flag.

2.3. Error handling and recovery

The parser reports syntax errors using a custom `ParseError` class, which captures the offending token, its location, and a descriptive message. When a parsing function encounters an unexpected token or invalid construct, it throws a `ParseError`, which is caught at the statement level (`statement()` method). This prevents the entire parsing process from failing on the first error.

To recover after an error, the parser uses a synchronization strategy inspired by panic-mode error recovery. Upon catching a `ParseError`, the parser enters a recovery mode via the `synchronize()` method. This skips tokens until it finds a likely boundary for the start of a new statement—typically identified by the `isAtStartOfLine` flag or the presence of keywords like var.

This approach allows the parser to continue parsing subsequent code and collect multiple errors in a single run. These errors are stored in an errors vector and can be reported all at once, improving the debugging experience for users writing or editing code.

3. Type Checking

This language implements a statically-typed system with support for classes, functions, and structured blocks. The type checker is a recursive post-order traversal algorithm that infers and verifies types using an environment-tracking approach, combined with local substitution of type variables and simple constraint solving.

3.1. Type Environments

The type checker maintains a stack of lexical environments, each mapping variables to optional types. A binding of `nullopt` represents a declared-but-not-defined variable, preventing illegal self-references during initialization.

```
using TypeEnv = std::unordered_map<VariableName,
std::optional<std::shared_ptr<Type>>>;
```

Variable scopes are pushed and popped explicitly with `beginScope()` and `endScope()`, while variables are introduced with `declare()` and bound to types with `define()`.

3.2. Local Type Inference

Type inference is implemented for variable declarations so that they do not have to be explicitly type annotated, such as:

```
var x = 1       // x: Int
var y = 2.0     // y: Double
var z = x + y   // Error: cannot add Int and Double
```

Function declarations, however, must be explicitly type annotated:

```
func add(x: Int, y: Int) -> Int {
  ...
}
```

3.3. Type Judgements

3.3.1. Expressions

$$\frac{}{\Gamma \vdash \text{Void}() : \text{Void}}\ [\text{T-Void}] \qquad \frac{}{\Gamma \vdash \text{n} : \text{Int}}\ [\text{T-Int}] \qquad \frac{}{\Gamma \vdash \text{d} : \text{Double}}\ [\text{T-Double}]$$

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}}\ [\text{T-True}] \qquad \frac{}{\Gamma \vdash \text{false} : \text{Bool}}\ [\text{T-False}] \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash \text{x} : \tau}\ [\text{T-Var}]$$

$$\frac{\Gamma \vdash f : (\tau_1 \times \tau_2, \ldots \times \tau_\text{n}) \to \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \ldots \quad \Gamma \vdash e_\text{n} : \tau_\text{n}}{\Gamma \vdash f(e_1, e_2, \ldots, e_\text{n}) : \tau}\ [\text{T-App}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \text{BinOp}(\otimes, \tau_1, \tau_1) = \tau_\text{r}}{\Gamma \vdash e_1 \otimes e_2 : \tau_\text{r}}\ [\text{T-BinOp}]$$

$$\frac{\Gamma \vdash e : \tau \quad \text{UnaryOp}(\ominus, \tau) = \tau_\text{r}}{\Gamma \vdash \ominus\, e : \tau_\text{r}}\ [\text{T-UnaryOp}]$$

$$\frac{\Gamma(x) = \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{x} = \text{e} : \tau}\ [\text{T-Assign}]$$

$$\frac{\text{self} \in \Gamma}{\Gamma \vdash \text{self} : \text{Instance(C)}}\ [\text{T-Self}]$$

$$\frac{\Gamma \vdash e : \text{Instance(C)} \quad \text{field}(C, x) = \tau}{\Gamma \vdash e.\dot{x} : \tau}\ [\text{T-Get}]$$

$$\frac{\Gamma \vdash e_1 : \text{Instance(C)} \quad \text{field}(C, x) = \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1.x := e_2 : \tau} \text{[T-Set]}$$

⊗	**Operand Type(s) t**	**Resulting Type t_r**
+, -, *, /	`Int`, `Double`	`t`
`%`	`Int`	`Int`
&&, ||	`Bool`	`Bool`
<, <=, >, >=	`Int`, `Double`	`Bool`
==, !=	`Int`, `Double`, `Bool`	`Bool`

Table 1: Binary operator types

⊖	**Operand Type t**	**Resulting Type t_r**
-	`Int`, `Double`	`t`
!	`Bool`	`Bool`

Table 2: Unary operator types

3.3.2. Statements

$$\frac{\Gamma[x \leftarrow \tau]\Gamma' \quad \Gamma' \vdash S : \tau'}{\Gamma \vdash x : \tau; S : \tau'} \text{[Sequence]}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma[x \leftarrow \tau] \vdash \text{var } x = e : \text{Void}} \text{[DeclareStmt]}$$

$$\frac{\Gamma[f \leftarrow (\tau_1 \times \tau_2 \times ... \times \tau_n) \to \tau_r]\Gamma' \quad \Gamma'[x_1 \leftarrow \tau_1, ..., x_n \leftarrow \tau_n] \vdash S : \text{return}(\tau_r)}{\Gamma \vdash \text{func } f(x_1{:}\tau_1,...,x_n{:}\tau_n) \text{ -> } \tau_r\{S\} : \text{Void}} \text{[FunctionStmt]}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e; : \text{Void}} \text{[ExprStmt]}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{return } e : \text{return}(\tau)} \text{[Return]}$$

$$\frac{\Gamma[C \leftarrow \text{ClassType}] = \Gamma_1 \quad \Gamma_1 \vdash \text{declarations} : \Gamma_2 \\ \Gamma_2 \vdash \text{method-sigs} : \Gamma_3 \quad \Gamma_3[\text{self} \leftarrow \text{Instance(C)}] \vdash \text{methods} : \text{Void}}{\Gamma \vdash \text{class } C\{ \text{ declarations ; methods } \} : \text{Void}} \text{[Class]}$$

$$\frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash S_1 : \text{return}(\tau) \quad \Gamma \vdash S_2 : \text{return}(\tau)}{\Gamma \vdash \text{if } (e) S_1 \text{ else } S_2 : \text{return}(\tau)} \text{[If]}$$

4. Bytecode

The Shiny compiler emits bytecode, which consist of instructions that the VM will execute to produce the program output. In this section, we specify the format of these instructions and a brief description of what each instruction does.

4.1. Specification

Each instruction in Shiny occupies 32-bits. The opcode occupies the first 8 bits, while the operand occupies the remaining 24 bits.

Name	Opcode	Stack (before)	Stack (after)	Action
NO_OP	0x00	- - -	- - -	
NIL	0x11	- - -	- - - A	A = nil
TRUE	0x12	- - -	- - - A	A = true
FALSE	0x13	- - -	- - - A	A = false
CONST	0x14	- - -	- - - A	A = constants[operand]
CLOSURE	0x15	- - -	- - - A	A = makeClosure(constants[operand])
ADD	0x31	- - - A B	- - - C	C = A + B
SUB	0x32	- - - A B	- - - C	C = A - B
MUL	0x33	- - - A B	- - - C	C = A * B
DIV	0x34	- - - A B	- - - C	C = A / B
MOD	0x35	- - - A B	- - - C	C = A % B
NEG	0x36	- - - A	- - - B	B = -A
EQ	0x37	- - - A B	- - - C	C = A == B
NEQ	0x38	- - - A B	- - - C	C = A != B
LT	0x39	- - - A B	- - - C	C = A < B
LTE	0x3a	- - - A B	- - - C	C = A <= B
GT	0x3b	- - - A B	- - - C	C = A > B
GTE	0x3c	- - - A B	- - - C	C = A >= B
AND	0x3d	- - - A B	- - - C	C = A && B
OR	0x3e	- - - A B	- - - C	C = A || B
NOT	0x3f	- - - A	- - - B	C = !A
BIT_AND	0x40	- - - A B	- - - C	C = A & B
BIT_OR	0x41	- - - A B	- - - C	C = A | B
BIT_XOR	0x42	- - - A B	- - - C	C = A ^ B
BIT_NOT	0x43	- - - A	- - - B	B = ~A
SHIFT_LEFT	0x44	- - - A B	- - - C	C = A << B
SHIFT_RIGHT	0x45	- - - A B	- - - C	C = A >> B
LOAD	0x50	- - -	- - - A	A = stack[bp + operand]
STORE	0x51	- - - A	- - -	stack[bp + operand] = A
DUP	0x52	- - - A	- - - A A	
POP	0x53	- - - A	- - -	
TEST	0x60	- - - A	- - -	if A then ip = ip + 1

Name	Opcode	Stack (before)	Stack (after)	Action
`JUMP`	`0x61`	`- - -`	`- - -`	`ip = operand`
`CALL`	`0x62`	`- - - A B C`	`- - - A B C`	`pushCallFrame()`
`RETURN`	`0x63`	`- - - A B C D`	`- - - D`	`popCallFrame(); D = A(B, C)`
`HALT`	`0x64`			
`GLOBAL_LOAD`	`0x70`	`- - -`	`- - - A`	`A = globals[operand]`
`GLOBAL_STORE`	`0x71`	`- - -`	`- - - A`	`globals[operand] = A`
`UPVALUE_LOAD`	`0x80`	`- - -`	`- - - A`	`A = upvalues[operand]`
`UPVALUE_STORE`	`0x81`	`- - - A`	`- - -`	`upvalues[operand] = A`
`UPVALUE_CLOSE`	`0x82`	`- - - A`	`- - -`	`closeUpvalue(A)`
`MEMBER_GET`	`0x90`	`- - - A`	`- - - B`	`B = A.members[operand]`
`MEMBER_SET`	`0x91`	`- - - A B`	`- - -`	`A.members[operand] = B`

Operation	Description
`makeClosure`	Iterates through the upvalue definitions in the `FunctionObject` loaded from `constants`, and captures the appropriate value on the stack into an open `UpvalueObject` for each upvalue definition.
`closeUpvalue`	Closes all upvalues in the upvalue stack up to the given upvalue.
`pushCallFrame`	Pushes the current closure/method, instruction pointer (ip), stack base pointer (bp) as a call frame onto the call stack.
`popCallFrame`	Pops a frame from the call stack and restores the closure/method, ip and bp from that frame. Also closes all upvalues and pops all but the top value from the stack, up to the pre-pop bp.

5. Memory Management

Swift uses reference counting for its memory management. It terms its implementation as Automatic Reference Counting (ARC), where memory management code is automatically inserted at compile-time to track references to each Swift object. This makes it different from garbage collection, which occurs at runtime and might incur some performance overhead. To prevent retain cycles, Swift allows for programmers to create **weak** or **unowned** references, which do not increment the reference count of an object.

In Shiny, we have implemented a custom reference counting algorithm, that currently supports only strong references. While the C++ Standard Template Library comes with a reference-counted smart pointer in `std::shared_ptr`, we are unable to directly use this because each `std::shared_ptr` requires 16-bytes of stack space, whereas our representation of values only has space for 8 bytes (described in Section 6). Therefore, we have implemented our own reference-counted smart pointers named `ObjectPtr`.

5.1. Implementation

To implement a custom smart pointer in C++, we need to create a class that wraps a raw pointer, and implement some reference-counting releated functionality when performing the following actions in a class' lifecycle:

1. Creation
2. Copying
3. Moving
4. Destruction

To support multiple types of Objects being pointed to, while still remaining type-safe, we leverage the C++ template language to define a generic type parameter `T`, which the `ObjectPtr` will point to. This leads to the following definition of the `ObjectPtr` and `Object` classes (with only the important methods):

```
template <typename T>
class ObjectPtr {
 public:
  ObjectPtr(T&& o);                     // creation
  ObjectPtr(const ObjectPtr& other);    // copying
  ObjectPtr(ObjectPtr&& other);         // moving
  ~ObjectPtr() noexcept(false);         // destruction
  T* get() const;
  T* operator->();

 private:
  Object* ptr;
};

class Object {
 public:
  template <typename T>
  Object(T&& o);
  ~Object() = default;

  template <typename T>
  T* get();

  std::variant</* object types */> data;
  int strongCount;
};
```

5.1.1. Creation

During the creation of an `ObjectPtr` using the `ObjectPtr(T&& o)` constructor, we allocate memory on the heap using `new Object`, which will store both the underlying object data and the `strongCount`. We initialize the `strongCount` of the `Object` to 1, as there is now exactly one pointer to the underlying `Object`: this newly created `ObjectPtr` instance.

5.1.2. Copying

When the `ObjectPtr` is copied, we increment the `strongCount`, as there is now one more pointer in existence pointing to the underlying `Object` than before. We can increment this strong count in the *copy constructor* `ObjectPtr(const ObjectPtr& other)`, which C++ conveniently allows us to override.

5.1.3. Moving

C++ also supports moving objects instead of copying them, during which we *do not* increment the `strongCount`. When a move occurs, the *move constructor* `ObjectPtr(ObjectPtr&& other)` is called instead. Within this constructor, we do not increment the strong count because we set the moved-from value to a `nullptr`, which means we have both created a new pointer and removed a pointer to the underlying `Object`.

5.1.4. Destruction

Finally, when an `ObjectPtr` is destroyed, we decrement the `strongCount`. If the strong count has reached 0, this means there are no remaining references to the underlying `Object`, so we call `delete ptr` to deallocate its memory.

5.2. Limitations

Unfortunately, we do not currently support weak or unowned references. This was due to the scope of Shiny's features rather than their viability within the current reference counting system.

To support weak references, a `weakCount` can be added to the `Object` class, which will track weak references rather than strong ones. Instead of deleting an `Object` when its strong count reaches 0, we delete both the strong and weak counts reach 0. However, a weak reference (which would be represented with a `WeakPtr` class) will return `std::nullopt` if we try to dereference it when its strong count has already reached 0, and we would decrement the weak count when this happens as well.

To support unowned references, an `UnownedPtr` class can be added that does not increment either the strong or weak counts, but is instead a simple pointer to the underlying `Object`.

6. Representing Values

In Swift, the way each value is represented in memory depends on the type of that value, with the exact layout specified by the Swift ABI. Type information in Swift is generally only present at compile-time, with the knowledge of how to manipulate values of a type encoded into the LLVM IR, and as a result, the machine code.

However, Shiny is more dynamic in nature, and requires that some type information be encoded into the runtime representation of values. Furthermore, we introduce **NaN-boxing** as a means of encoding this information, while maintaining a size of 8 bytes per value.

6.1. NaN-boxing

We have used the NaN-boxing technique to represent both primitive data types (booleans, integers, floats) and complex data types (pointers to objects) within the same 8-byte memory space, while maintaining the knowledge of which type of data is contained. This is achieved by exploiting 48 unused bits of a floating-point NaN (Not a Number) value, using masks and bitwise operations to distinguish between an intended NaN value and another type of data stored within the unused bits. The technique allows us to cut the memory usage of values in half, since the alternative would be storing data and type information in 16 bytes.

7. Representing Closures

Closures are an important feature in Swift, used for callbacks, completion handlers in asynchronous code, functional programming, etc. Shiny also has basic support for closures, specfically the capturing of variables in the surrounding environment in which a closure is defined, and also closures that escape the environment they are defined in. To achieve this, we use a similar solution to the Lua programming language known as *upvalues*.

7.1. Upvalue Implementation

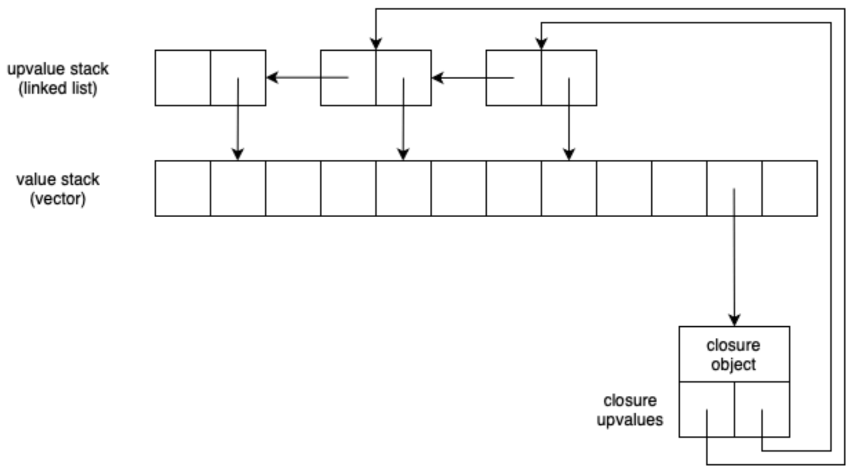

Figure 3: Example of a closure object at runtime

As shown in Figure 3, alongside the stack of values used for main operations, the VM also maintains a stack of upvalues, implemented as a linked list of `UpvalueObjects`. A single `UpvalueObject` consists of a pointer to the next object in the upvalue stack, and a pointer to the value it captures. Each `ClosureObject` stores a list of `UpvalueObjects` too, and these point to the same upvalues on the stack.

Currently, all the upvalues in Figure 3 point to values that still exist on the stack, and are thus in the *open* state. However, if the function responsible for these upvalues returns, the values that these upvalues point to will be invalidated. Thus, we have to first *close* these upvalues by moving the values they point to to the heap, as shown in Figure 4.

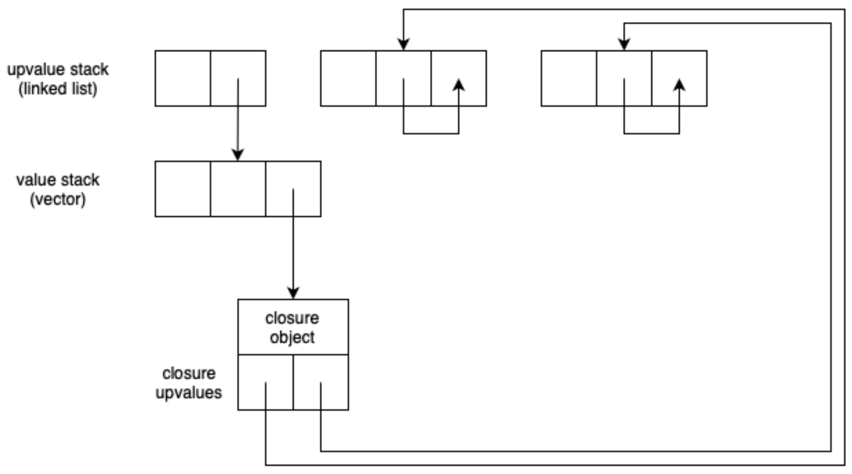

Figure 4: Example of a closure object containing upvalues in closed state

This allows for a behaviour similar to the implementations covered in class, but improves upon it by capturing only the variables that are used, rather than the entire environment frame. This is useful especially in the context of Swift, where reference cycles can be created by capturing a variable that the programmer did not intend to capture.

8. Building and Testing

Our project requires a reasonbly recent version of CMake and C++ to be installed. To build the project, the following commands need to be run:

```
cmake -B build/        # establishes the build/ folder
cmake --build build/   # builds the project
```

To run Shiny in REPL (interactive) mode, simply run the built executable without any arguments:

```
build/src/shiny
```

To execute a Shiny file, simply run the built executable with the filename as the argument.

```
build/src/shiny file.swift
```

To run the tests (include the end-to-end tests), simply run the built tests executable.

```
build/tests/tests
```

Should all the tests pass, the following output should be shown:

```
[==========] 42 tests from 4 test suites ran. (12 ms total)
[  PASSED  ] 42 tests.
```

9. Limitations and Future Work

Shiny currently supports only a small subset of Swift features. In particular, some of the features that might make Shiny more complete include supporting weak and unowned references, generics in type checking (which would open up support for arrays, dictionaries and optionals), and class inheritance. There are also a couple of areas we could improve on, including better error messages, basic optimisations for VM bytecode.