



MoonRust

Memory Safe Lua
Interpreter

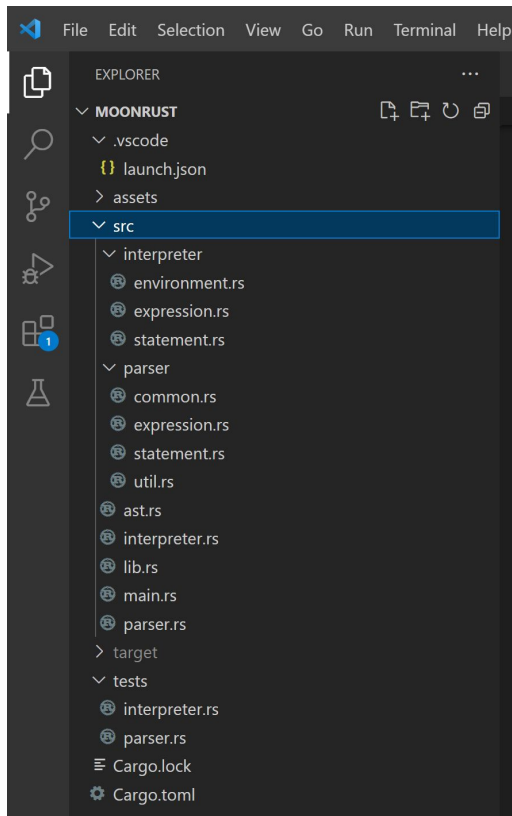
Project Description

- Goal: Build an interpreter to execute a subset of Lua from a file
- MVP Features:
 - Parser (excluding some syntactic sugars)
 - Variable assignments
 - Binary expression
 - Unary expressions
 - Control statement evaluation (if, else, break)
 - Loop statement evaluation (for, while, repeat - excluding for generics)
 - Function definition/call
 - Visibility rules and scoping
 - Table evaluation
 - Some standard library functions (print, read, random)

Demo



Code Structure



- Entry Point
 - main.rs
 - Program text and command line args
- Parser
 - Input file → AST
 - Separate parsing files for statement & expression
- Interpreter
 - Evaluation
 - Environment represents scope

Parser



- What: Converts the input file into an Abstract Syntax Tree
- How: Nom parser combinator library
- Our main challenge: The Lua manual's specified syntax grammar is left recursive
 - Why: Nom is a top-down parser
 - How we solved this issue: factor out left recursion
 - Parse expressions according to the specified operator precedence
 - Flatten rules into one unambiguous rule

`prefixexp ::= var | functioncall | '(' exp ')'`

`functioncall ::= prefixexp args | prefixexp ':' Name args`

AST

```
pub enum Statement {  
    Assignment((Vec<Var>, Vec<Expression>, bool)), // bool flag: true if  
    FunctionCall(FunctionCall),  
    Break,  
    DoBlock(Block),  
    While((Expression, Block)),  
    Repeat((Block, Expression)),  
    If((Expression, Block, Vec<(Expression, Block)>, Option<Block>)),  
    ForNum((String, Expression, Expression, Option<Expression>, Block)),  
    ForGeneric((Vec<String>, Vec<Expression>, Block)),  
    FunctionDecl((String, ParList, Block)),  
    LocalFuncDecl((String, ParList, Block)),  
    Semicolon,  
}
```

```
pub struct Block {  
    pub statements: Vec<Statement>,  
    pub return_stat: Option<Vec<Expression>>,  
}
```

```
pub enum Expression {  
    Nil,  
    False,  
    True,  
    Numeral(Numeral),  
    LiteralString(String),  
    DotDotDot, // Used for a variable number of arguments  
    FunctionDef((ParList, Block)),  
    PrefixExp(Box<PrefixExp>),  
    TableConstructor(Vec<Field>),  
    BinaryOp((Box<Expression>, BinOp, Box<Expression>)),  
    UnaryOp((UnOp, Box<Expression>)),  
}
```

Interpreter

```
impl Expression {  
    pub fn eval<'a, 'b>(&'a self, env: &'b mut Env<'a>) -> Result<Vec<LuaValue<'a>>, ASTExecError> {
```

```
impl Statement {  
    pub fn exec<'a, 'b>(  
        &'a self,  
        env: &'b mut Env<'a>,  
        ) -> Result<Option<Vec<LuaValue>>, ASTExecError> {
```

Interpreter

```
pub struct LuaValue<'a>(Rc<LuaVal<'a>>);
```

```
pub enum LuaVal<'a> {  
    LuaTable(LuaTable<'a>),  
    LuaNil,  
    LuaBool(bool),  
    LuaNum([u8; 8], bool), // numerals as an array of 8 bytes, bool for is_float  
    LuaString(String),  
    Function(LuaFunction<'a>),  
    Print,  
    TestPrint(Rc<RefCell<Vec<String>>>),  
    Read,  
    Random,  
}
```


Environment

```
pub struct EnvTable<'a>(Rc<RefCell<HashMap<String, LuaValue<'a>>>>);
```

```
pub struct LocalEnv<'a>(Vec<Option<EnvTable<'a>>>);
```

```
pub struct Env<'a> {  
    global: EnvTable<'a>,  
    local: LocalEnv<'a>,  
}
```

Environment

- New EnvTable is added when exiting the “Block”
- For the example code, “a” and “b” will be in the same scope, but they will be stored in different “EnvTable”
- Output “1” and “nil”
- Eg. [None, EnvTable1, EnvTable2, None, EnvTable3]
- It behaves like Lua!

```
local a = 1
function g()
    print(a)
    print(b)
end
local b = 3
g()
```

Rusty Code

- Match Expressions for Enums
 - Covered all possible variants
- Use of Rc and RefCell
 - Multiple owners and mutability
- Display trait
 - Verify AST parsing
 - Debug LuaValue

```
pub struct EnvTable<'a>(Rc<RefCell<HashMap<String, LuaValue<'a>>>>);

pub struct LuaValue<'a>(Rc<LuaVal<'a>>);

pub struct LuaTable<'a>(RefCell<HashMap<TableKey, LuaValue<'a>>>)
```

Difficult in Rust

- Environment Implementation
 - Rcs and RefCells
 - `Vec<(String, LuaValue<'a>> → HashMap<String, LuaValue<'a>>`
- Function Default Parameters
 - `Block::exec` and `Block::exec_without_pop` vs boolean flag

Testing

- 92 unit tests and 33 integrations tests
- Verify edge cases

```
$ cargo test -q

running 92 tests
..... 88/92
....
test result: ok. 92 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.15s


running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s


running 33 tests
.....
test result: ok. 33 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.03s


running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Execution Time

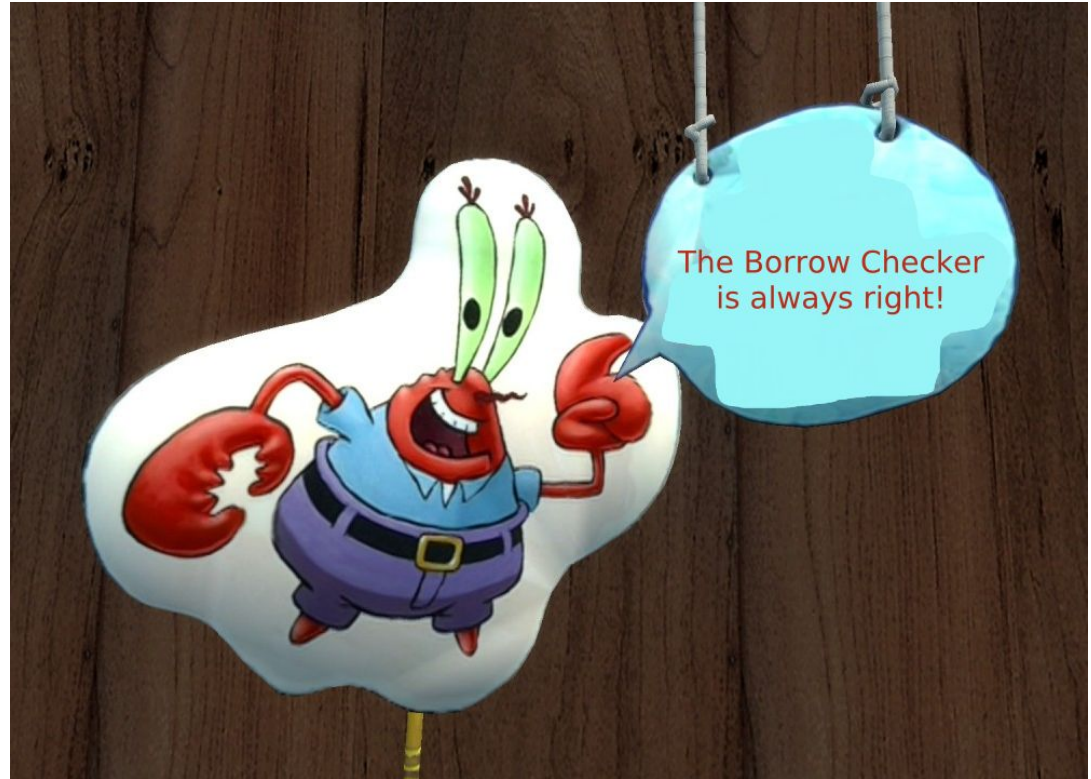
- Prime number test ($n = 4829449$)
 - Official Lua: < 1 second
 - MoonRust: ~34 seconds
- Fibonacci numbers 1 to 30
 - Official Lua: ~1 second
 - MoonRust: ~163 seconds

Conclusion

- Challenging project, but finished MVP
- Multiple design changes
- Using Rust helped to write readable code and to catch easy-to-miss errors
- A lot of optimizations are needed in the interpreter
- Acknowledgment: Dr. Fluet for almost being a fourth member of the team

Thank you!

Any Questions?



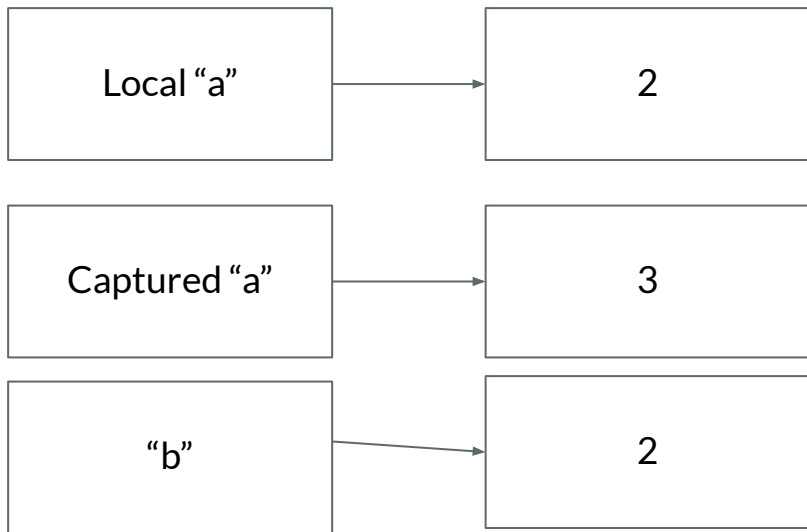
Extra slides

Capturing Variables

- A Lua function can act like a closure
- Well, LuaValue is Rc, so we can iterate through closure body and clone the Rc
- Little bit of overhead for defining, but Rc clone should be cheap!
- Create “capture_variable” function for all types in AST
- However,

Capturing Variables

```
local a = 2
function f()
    a = a + 1
end
b = a
f()
print(a)
print(b)
```



- Should output 3 and 2, but output 2 and 2
- Captured "a" and local "a" will be pointing to the different values after re-assignment!
- Using "RefCell" will also not solve the problem

Environment

```
pub struct EnvTable<'a>(HashMap<String, LuaValue<'a>>);

pub struct LocalEnv<'a>(Vec<EnvTable<'a>>);

pub struct Env<'a> {
    global: EnvTable<'a>,
    local: LocalEnv<'a>,
}
```

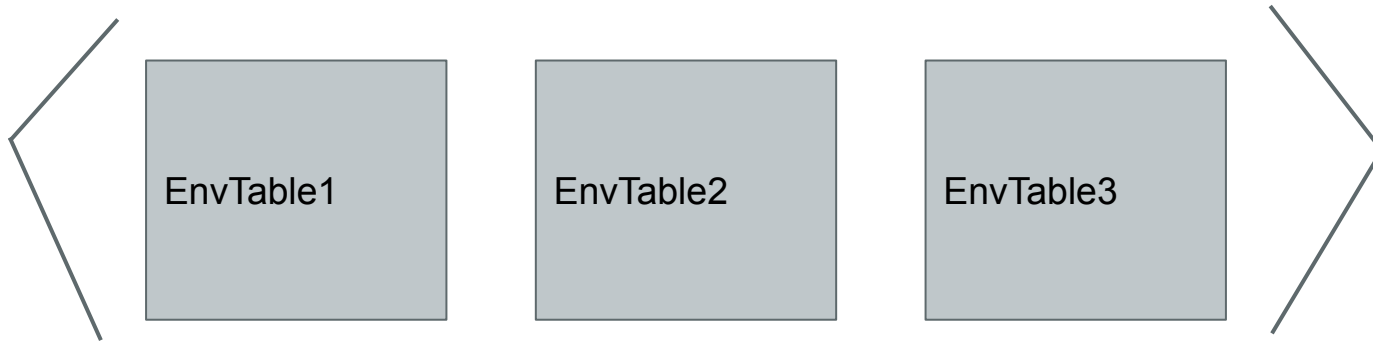
Environment

```
pub struct EnvTable<'a>(Rc<RefCell<HashMap<String, LuaValue<'a>>>>);  
  
pub struct LocalEnv<'a>(Vec<EnvTable<'a>>);  
  
pub struct Env<'a> {  
    global: EnvTable<'a>,  
    local: LocalEnv<'a>,  
}
```

- Capture the scope when closure is defined!

Environment

Each EnvTable represents **scope** in local environment



Environment

```
local a = 1
function g()
    print(a)
    print(b)
end
local b = 3
g()
```

- “b” is added to the scope after closure is defined!
- Need to differentiate variables captured and not captured in the same scope