Lua

By James Epstein and Joseph Harwood

Introduction to Lua

- A first look at Lua might resemble python
- The syntax is close to a readable language
- It is dynamically typed
- It is interpreted and uses garbage collection
- It is commonly used in AI pathing

Lua Syntax: Variables

Types:

1. Number
2. String
3. Boolean
4. Nil
5. Table
6. Function
7. Thread
8. UserData

Lua variables are polymorphic

```lua
-- Number:
-- Numbers are held as double-precision floating point values
local num = 777
num = num/2
print("777/2: "..num) -- prints as a float
num = num*2
print("777: "..num) -- prints as an int

-- String:
-- Strings consist of any number of 8-bit characters
local str = "Hello World"
print(str..num) -- strings can concatenate with numbers, but not other types

-- Boolean:
-- Lua treats "false" and "nil" as false and everything else to be true
local bool = true
print(false == 0) -- false
print(true == nil) -- false

-- Nil:
-- Nil represents the absence of a value in Lua
local none = nil

-- Table:
-- Tables in Lua are associative arrays, and fill the role of both arrays and dictionaries
-- Note: Table variables hold references to their tables
-- Note: Arrays in Lua traditionally start with index 1
local tab = {}
tab["a"] = 1
print(tab)

-- Function:
-- Function data is used since Lua can manipulate Lua and C functions
-- Note: Function variable hold references to their functions
function adder(a,b)
 return a+b
end
local adder2 = adder
print(adder)
print(adder2)

-- Userdata:
-- Userdata is used to hold random C data for Lua manipulation

-- Thread:
-- Thread is used for multithreading
```

Lua Syntax: Functions

```lua
-- Functions in Lua can be treated like any other data type
-- Functions can be assigned to variables and passed into functions as parameters
function printer(n)
 print(n)
end

local func = printer

print(func)

-- The function can be executed by using the variable name
func("hi")
```

Functions in lua are treated as a regular data type

Parts:

1. Function declaration
2. Body of the function
3. Return statement
4. End statement

Lua Syntax: Statements

```lua
if x > 0 then
    print("x is greater than 0")
elseif x < 0 then
    print("x is less than 0")
else
    print("x is 0")
end

while x < 10 do
    x = x + 1
end

for var = 0, 10, 1 do
    print("var: " .. var)
end

repeat
    x = x - 1
until x == 0
```

- In Lua, most logic statements must end with an "end" command
- Lua if statements use "if ... then", "elseif" and "else" as keywords
- Lua for loops can loop over any set of values
- Lua loops use a "keyword [expression] do" syntax

Lua Syntax: Tables

```lua
local testArray = {}                    -- Local table declaration, lua doesn't have traditional arrays
testArray[1] = 4                        -- It is traditional to start lua "arrays" at index 1
for i = 2,8 do
    testArray[i] = i + 2                -- Populate the lua "array"
end
testArray[9] = 10
testArray[10] = 55
for i,val in ipairs(testArray) do       -- Loop through the key, value pairs in testArray
    print("testArray["..i.."] = "..val)
end
-- testArray should now be {1:4,2:4,3:5,4:6,5:7,6:8,7:9,8:10,9:10,10:55}
```

- Lua does not have an array type, only a table type
- Tables in Lua resemble Python dictionaries
- Lua "arrays" are tables with sequential, integer keys
- Lua "arrays" conventionally start with 1
- The Lua function "ipairs()" returns the numeric key-value pairs in increasing order
- The Lua function "pairs()" returns all key-value pairs in no set order

Lua Objects

- Classes are defined by associating a metatable with methods to a class instance
- Classes in Lua can be dynamically redefined and changed at runtime

```
-- Classes are supported in lua, but rather oddly defined
Mather = {}
Mather.__index = Mather

function Mather:new(a,b)
 local math = {}
 setmetatable(math,Mather)
 math["first"] = a
 math["second"] = b
 return math
end

function Mather:add()
 return (self.first + self.second)
end

function Mather:subtract()
 return (self.first - self.second)
end

math = Mather:new(10,5)
print("10+5: "..math:add())
print("10-5: "..math:subtract())
```

Lua Multiple Assignment

```lua
function noReturner()
    return
end

function oneReturner()
    local a = 9001
    return a
end

function fourReturner()
    local a, b, str, bool = 9001, 9002, "A super interesting string", false
    return a, b, str, bool
end

-- In lua, the number of variables and the number of values do not need to match
-- Extra variables get nil and extra values are discarded
a, b, c = noReturner()
print(a, b, c)      -- nil, nil, nil
a, b, c = oneReturner()
print(a, b, c)      -- 9001, nil, nil
a, b, c = fourReturner()
print(a, b, c)      -- 9001, 9002, A super interesting string

-- Function returns can be included with standard assignments
-- A Functions returns will be assigned to any additional variables in the assignment
a, b, c = 42, oneReturner(), 6
print(a, b, c)      -- 42, 9001, 6
a, b, c = 42, fourReturner()
print(a, b, c)      -- 42, 9001, 9002

-- If a function with multiple returns is followed by a standard value in an assignment,
-- only the first value is assigned even if this results in variables being assigned nil
a, b, c, d = 42, fourReturner(), 6
print(a, b, c, d)       -- 42, 9001, 6, nil
a, b, c, d = 42, fourReturner()
print(a, b, c, d)       -- 42, 9001, 9002, A super interesting string
a, b, c, d = fourReturner(), 6
print(a, b, c, d)       -- 9001, 6, nil, nil
a, b, c, d = 42, 6, fourReturner()
print(a, b, c, d)       -- 42, 6, 9001, 9002
```

Lua Overloading

```lua
--method that handles the behavior of the '+' operator when linked lists are involved
--append, push, or concatenate based on the types received
function LinkedList:__add(other)

    --testing to make sure each one is type table
    if type(other) == "table" and type(self) == "table" then

        --making sure that each table is of the linked list class
        if other.__index == LinkedList and self.__index == LinkedList then

            --set next field of the tail to the head of the second list
            --set the new tail
            self.tail.next = other.head
            self.tail = other.tail

            --increment the length of the left list by the length of the right list
            self.length = self.length + other.length

            return self
        end
    end

    --testing to see if the left value is a table
    if other ~= nil and type(self) == "table" then

        --if self is a linked list then append other as a data value
        if self.__index == LinkedList then
            self:append(other)
            return self
        end
    end

    --testing to see if the right value is a table
    if self ~= nil and type(other) == "table" then

        --if other is a linked list then push self as a data value
        if other.__index == LinkedList then
            other:push(self)
            return other
        end
    end

    return nil

end
```