# Lua

Aashish Lalani

CIS554

## Why Lua?

- Lightweight
  - Can run on pretty much any specifications. Tradeoff is very few provided functions
- Embedability
  - Scripting language which works on almost any hardware (iPhone, Android, PS3)
  - An API is provided that allow use in C/C++, Java, C#, etc
- Simple
  - Easy to read, fast performance. Tradeoff is complexity, better algorithms need to be implemented

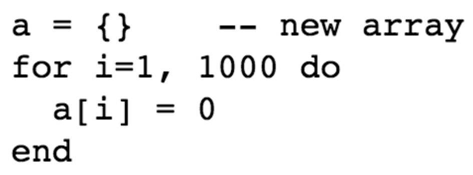
## Basics

- Offers the familiar datatypes: Numbers, String, Booleans
- Expressions: Arithmetic (+ - * /) Relational (< > == ~=) Logical (and or not) Concatenation ( .. )
- Control Structures: if then else, while, for, etc
- Functions
- Iterators
- Data Structures: Arrays, LLs, Queues, Sets, etc but 1 big difference

## There is actually only 1 data structure: Table

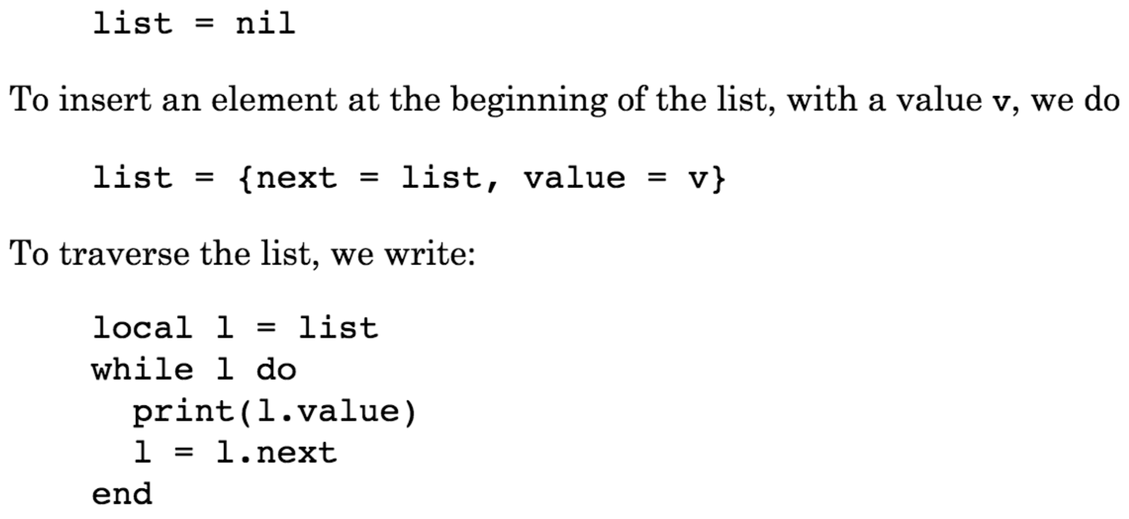
Tables can be used to implement all other data structures

- Example:
  - Array:



```
a = {}    -- new array
for i=1, 1000 do
  a[i] = 0
end
```

  - LL



```
list = nil
```

To insert an element at the beginning of the list, with a value **v**, we do

```
list = {next = list, value = v}
```

To traverse the list, we write:

```
local l = list
while l do
  print(l.value)
  l = l.next
end
```

## Metatable and Metamethods

- Help you change how specific tables behave.
  - Example: Change behaviour of + to set union for sets
  - Set = {}

```
function Set.union (a,b)
  local res = Set.new{}
  for k in pairs(a) do res[k] = true end
  for k in pairs(b) do res[k] = true end
  return res
end
```

  - Create metatable for sets

```
Set.mt = {}    -- metatable for sets
```

- Make set constructor function that makes all the sets' metatable equal to the same metatable:

```
function Set.new (t)   -- 2nd version
  local set = {}
  setmetatable(set, Set.mt)
  for _, l in ipairs(t) do set[l] = true end
  return set
end
```

- Implement metamethod:

```
Set.mt.__add = Set.union
```

## Using metatables to implement OOP

```
Account = {balance = 0}

function Account:new (o)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  return o
end

function Account:deposit (v)
  self.balance = self.balance + v
end

function Account:withdraw (v)
  if v > self.balance then error"insufficient funds" end
  self.balance = self.balance - v
end
```

## What if we wanted a subclass?

```
SpecialAccount = Account:new()
s = SpecialAccount:new{limit=1000.00}
```

When new executes now, self will refer to a SpecialAccount.

If we run s:deposit(100.00) it will try to access SpecialAccount’s deposit, realize it doesn’t exist, and then default to Account’s deposit function. Can also override functions:

```
function SpecialAccount:withdraw (v)
  if v - self.balance >= self:getLimit() then
    error"insufficient funds"
  end
  self.balance = self.balance - v
end

function SpecialAccount:getLimit ()
  return self.limit or 0
end
```

## References and further reading

- https://www.lua.org/pil/contents.html
- ZeroBrane IDE for Lua