Lua + Löve

Lua

- mais uma linguagem dinâmica
  - alguma similaridade com Python, Perl, e outras
- linguagem de descrição de dados
- ênfase em scripting
  - comunicação inter-linguagens

dinamismo

- tipagem dinâmica
  - verificações em tempo de execução: valores têm tipo
  - tipagem forte: linguagem não aplica operações a tipos incorretos
- interpretação dinâmica de código
  - linguagem capaz de interpretar código dinamicamente no mesmo ambiente de execução do programa

como usar Lua

- *stand alone* X embutida em algum programa

- www.lua.org

usando o interpretador

```
$ lua -e "print(2^0.5)"
```

```
$ lua nome-do-arquivo
```

```
$ lua
> print(2^0.5)
```

```
$ lua
> dofile("nome-do-arquivo")
```

tipos

- number
- string
- boolean
- nil
- function
- table
- thread
- userdata

number

- double
- int

Boolean

- sem exclusividade em testes
- operadores booleanos operam sobre todos os tipos
- `nil` e `false` testam como negativos

```
print (0 or 6)
print (nil or 10)
print (x or 1)
```

exemplos

soma elementos array

```
function add (a)
  local sum = 0
  for i = 1, #a do
    sum = sum + a[i]
  end
  return sum
end

print(add({10, 20, 30.5, -9.8}))
```

soma elementos array

```
function add (a)
  local sum = 0
  for i = 1, #a do
    sum = sum + a[i]
  end
  return sum
end

print(add({10, 20, 30.5, -9.8}))
```

soma elementos array

```
function add (a)
  local sum = 0
  for i = 1, #a do
    sum = sum + a[i]
  end
  return sum
end

print(add({10, 20, 30.5, -9.8}))
```

soma linhas de arquivo

```
function addfile (filename
  local sum = 0
  for line in io.lines(filename) do
    sum = sum + tonumber(line)
  end
  return sum
end
```

io

io.read – padrões:
"*a" reads the whole file
"*l" reads the next line (without newline)
"*L" reads the next line (with newline)
"*n" reads a number
num reads a string with up to *num* characters


Funções em Lua

- funções são valores dinâmicos de primeira classe

```
(function (a, b) print (a+b) end) (10, 20)
```

```
table.sort(t, function (a,b)
                return a.key < b.key
              end))
```

Funções e variáveis

- usamos a sinaxe convencional para armazenar fçs em variáveis

```
inc = function (a) return a+1 end
```

```
function inc (a)
  return a+1
end
```

Múltiplos retornos

- funções em Lua podem retornar múltiplos valores
- atribuição múltipla e ajuste de valores

Escopo

- variáveis locais e globais

```
local a = 5 print(a) --> 5
do
  local a = 6 -- create a new local inside the do block
  print(a) --> 6
end
print(a) --> 5
```

Escopo léxico e closures

```
local function f()
  local v = 0
  return function ()
    local val = v
    v = v+1
    return val
  end
end

cont1, cont2 = f(), f()
print(cont1())
print(cont1())
print(cont2())
print(cont1())
print(cont1())
print(cont2())
```

Regiões geométricas

- a função abaixo cria regiões circulares:

```
function circle (cx, cy, r)
  return function (x, y)
    return (x - cx)^2 + (y - cy)^2 <= r^2
  end
end
```

```
c1 = circle(5.0, -3.2, 4.5)
c2 = circle(0, 0, 1)
```

Combinando regiões

```
function union (r1, r2)
  return function (x, y)
    return r1(x, y) or r2(x, y)
  end
end
```

```
function inter (r1, r2)
  return function (x, y)
    return r1(x, y) and r2(x, y)
  end
end
```

Tabelas

- única forma de estruturar dados em Lua
- arrays, structs, estruturas de dados, ...
- "arrays associativos": podem ser indexados por valores de quaisquer tipos

Construtores

- criação e inicialização de tabelas

```
{}
{x = 5, y = 10}
{“Sun”, “Mon”, “Tue”}
{[exp1] = exp2, [exp3] = exp4}
```

Todos os prefixos de uma string

```
function prefixes (s, len)
  len = len or 0
  if len <= #s then
    return string.sub(s, 1, len),
           prefixes(s, len + 1)
  end
end
```

```
print(prefixes("alo"))  -->   a    al    alo
t = {prefixes("vazavaza")}
```

Estruturas de Dados (2)

- Arrays: inteiros como índices

```
a = {}
for i=1,n do a[i] = 0 end
print(#a)
```

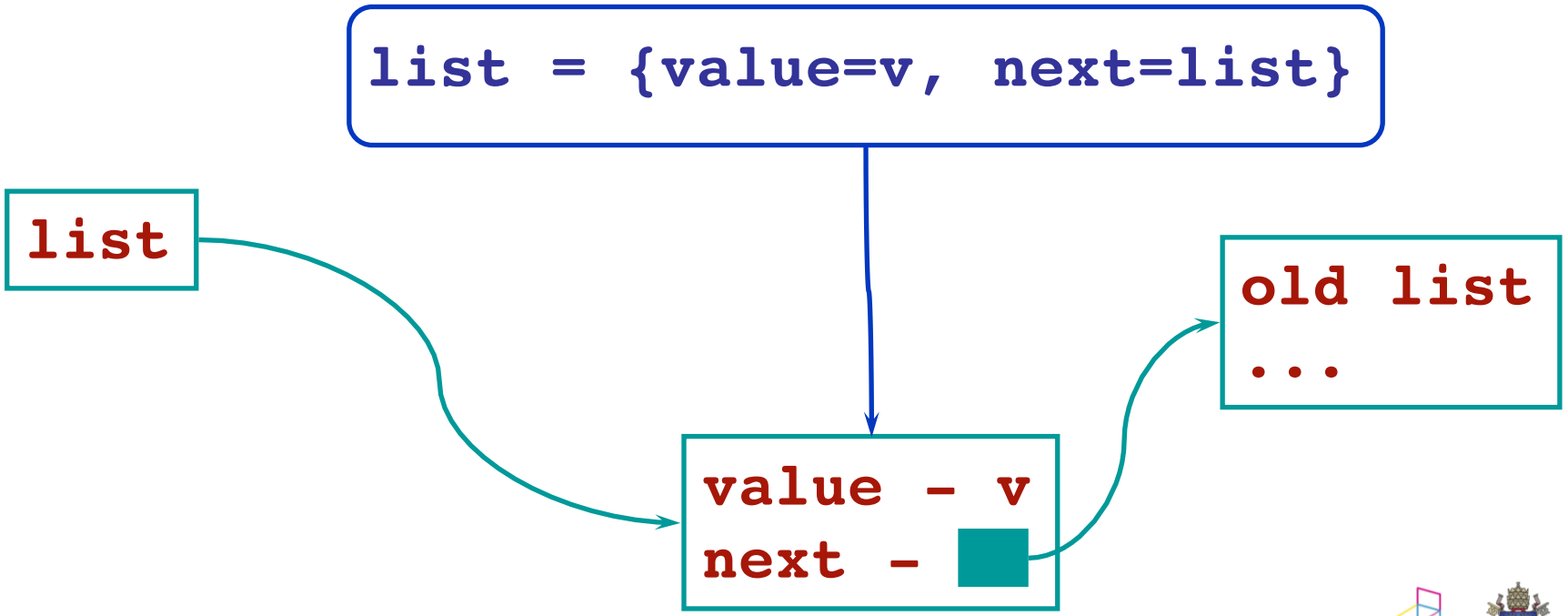
- Conjuntos: elementos como índices

```
t = {}
t[x] = true       -- t = t ∪ {x}
if t[x] then      -- x ∈ t?
  ...
```

PUC RIO

Listas Encadeadas

- Tabelas são *objetos*, criados dinamicamente



percurso de tabelas

- arrays: ipairs
- outras tabelas: pairs

```
nt = {}
for k, v in pairs(t) do
  nt[v] = k
end
```

Objetos

- funções de 1ª classe + tabelas ≈ objetos

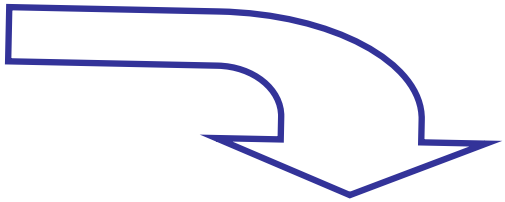
```
Rectangle = {w = 100, h = 250}

function Rectangle.area ()
  return Rectangle.w * Rectangle.h
end
```

```
function Rectangle.area (self)
  return self.w * self.h
end
```

Chamada de métodos

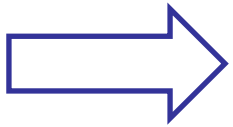
- açucar sintático para métodos
  - cuida de *self*

```
function a:foo (x)
  ...
end
```



```
a.foo = function (self,x)
  ...
end
```

```
a:foo(x)
```



```
a.foo(a,x)
```

löve

- framework para jogos 2D
- Linux, Windows, MacOS, Android, iOS
- love2d.org

löve – estrutura do jogo

- love.load
- love.update
- love.draw

löve: exemplo bobo

```
function love.load()
  text = "hello!"
  image = love.graphics.newImage( "alter.jpg" )
  x = 50 y = 200
end

function love.update (dt)
  x = x + 50*dt
  if x > 300 then x=50 end
end

function love.draw ()
  love.graphics.draw(image, x, y, 0, 0.1, 0.1)
  love.graphics.print(text, x, y-50)
end
```

interação com eventos

- testes no loop como no arduino
  - testes:

```
function love.update (dt)
  x = x + 50*dt
  if love.keyboard.isDown("down") then
    y = y + 10
  end
end
```

- mas também callbacks para eventos *discretos*

```
love.keypressed (key)
```

löve: interação

```
function love.load()
  x = 50 y = 200
  w = 200 h = 150
end

function naimagem (mx, my, x, y)
  return (mx>x) and (mx<x+w) and (my>y) and (my<y+h)
end

function love.keypressed(key)
  local mx, my = love.mouse.getPosition()
  if key == 'b' and naimagem (mx,my, x, y) then
     y = 200
  end
end

function love.update (dt)
  ...
```

löve: interação

```
function love.load()
  x = 50 y = 200 w = 200 h = 150
end

...
function love.keypressed(key)
  ...
end

function love.update (dt)
  local mx, my = love.mouse.getPosition()
  if love.keyboard.isDown("down") and naimagem(mx, my, x, y) ther
    y = y + 10
  end
end

function love.draw ()
  love.graphics.rectangle("line", x, y, w, h)
end
```

callbacks e estado

```
function love.keypressed(key)
 local mx, my = love.mouse.getPosition()
  if key == 'b' and naimagem (mx,my, x, y) then
     y = 200
  end
end
function love.update (dt)
  local mx, my = love.mouse.getPosition()
  if love.keyboard.isDown("down") and naimagem(mx, my, x, y) then
    y = y + 10
  end
end
function love.draw ()
  love.graphics.rectangle("line", x, y, w, h)
end
```

- variáveis globais capturam estado como no Arduino
- mas podemos usar características da linguagem para encapsular esse estado

encapsulando estado

```
function retangulo (x,y,w,h)
  local originalx, originaly, rx, ry, rw, rh =
                                  x, y, x, y, w, h
  return {
    draw =
      function ()
        love.graphics.rectangle("line", rx, ry, rw, rh)
      end,
    keypressed =
      function (key)
        local mx, my = love.mouse.getPosition()
        ...
      end
  }
end
...
function love.load()
  ret1 = retangulo (50, 200, 200, 150);
end
```

encapsulando estado

```
function retangulo (x,y,w,h)
  local originalx, originaly, rx, ry, rw, rh =
                                  x, y, x, y, w, h
  return {
    draw =
      function ()
        love.graphics.rectangle("line", rx, ry, rw,
rh)
      end,
    keypressed =
      function (key)
        local mx, my = love.mouse.getPosition()
        ...
      end
  }
end
function love.load()
  ret1 = retangulo (50, 200, 200, 150);
end
```

encapsulando estado

```
function love.load()
  ret1 = retangulo (50, 200, 200, 150);
end

function love.keypressed(key)
  ret1.keypressed(key)
end
```

exercício

1. abrir o ZeroBrane
   - escolher "project"-> "lua interpreter"-> löve
2. executar código em
   - sr-19/code/lovelua/retangulo1/main.lua

   ("executar" o diretorio que contém a main.lua)
3. retirar chamada a `love.keyboard.isDown` e programar a reação à tecla "`down`" dentro de `keypressed` – ver o que muda!
4. incluir reação a tecla `"right"`, andando com retângulo para a direita
5. encapsular o retângulo como indicado em slides "encapsulando estado" – Agora keypressed e update devem chamar chamar fçs "keypressed" e "update" do retângulo
6. criar dois retângulos em sua aplicação, em posições diferentes
7. criar um array de retângulos