

# *Estruturas de Linguagem*

## *Closures & Co-rotinas*

**Francisco Sant'Anna**

**francisco@ime.uerj.br**

**<http://github.com/fsantanna-uerj/EDL>**

# Abstração de Código (definida pelo programador)

- “Subprogramas” definidos pelo programador
- Abstração para com sequência de instruções
- Detalhes desnecessários são escondidos
- O programa “chama” o subprograma, passando-o temporariamente o controle da CPU
- Economia de memória e tempo de desenvolvimento

# Características Gerais

- Um único ponto de entrada
- O chamador é suspenso enquanto o subprograma chamado executa
  - implica que só há um subprograma em execução
- O controle retorna ao chamador ao fim da execução
- Threads, Corrotinas, etc?

# Parâmetros

- Como “configurar” o subprograma?
  - acesso direto à não locais (globais, `upvals`, campos)
  - passagem de parâmetros
    - formal paramters vs actual parameters (arguments)
    - por posição ou por chave



# Closures

Defining a closure is a simple matter; a **closure** is a **subprogram** and the referencing **environment** where it was defined. The referencing environment is needed if the subprogram can be **called from any arbitrary place** in the program. Explaining a closure is not so simple.

```
function f1()  
  local x = 10  
  local function f2()  
    return x  
  end  
  return f2  
end  
  
local f = f1()  
print( f() )
```

- Só fazem sentido quando
  - subprogramas podem ser aninhados
  - chamadas com ambientes originais fora de escopo
- Ambiente é capturados e movido da pilha para a heap.

# Closures

- Closure = função + ambiente
- Função
  - protótipo, estático
- Ambiente
  - variáveis livres
  - registro, dinâmico

# Closures

- Relação com objetos

```
function new (x, y)
  return {
    move = function (dx,dy)
      x = x + dx
      y = y + dy
      return x, o.y
    end

    ...
  }
end
local o1 = new(0,0)
local o2 = new(100,100)
print( o1.move(10,10) )
print( o2.move(20,20) )
print( o1.move(-5,-5) )
print( o2.move(-5,-5) )
```





# Características Gerais

- Um único ponto de entrada
- O chamador é suspenso enquanto o subprograma chamado executa
  - implica que só há um subprograma em execução
- O controle retorna ao chamador ao fim da execução
- Threads, Corrotinas, etc?

# Co-routines

Coroutines can have **multiple entry points**, which are **controlled** by the coroutines **themselves**. They also have the means to **maintain their status** between activations. This means that coroutines must be **history sensitive** and thus have static **local variables**. Secondary executions of a coroutine often begin at **points other than its beginning**. Because of this, the invocation of a coroutine is called a **resume** rather than a call.

```
r1 = resume c1(10)
print("fora", r1)
r2 = resume c1(20)
print("fora", r2)
r3 = resume c1(30)
```

```
coro c1(v1)
  print("dentro", v1)
  local v2 = yield(v1)
  print("dentro", v1+v2)
  return v1+v2
end
```

?

# Co-rotinas

## Simétricas vs Assimétricas

```
r1 = resume c1(10)
print("fora", r1)
r2 = resume c1(20)
print("fora", r2)
r3 = resume c1(30)
```

```
coro c1(v1)
  print("dentro", v1)
  local v2 = yield(v1)
  print("dentro", v1+v2)
  return v1+v2
end
```

```
coro c1(v1)
  print("dentro", v1)
  local v2 = resume main(v1)
  print("dentro", v1+v2)
  resume main(v1+v2)
end
```

# Co-rotinas em Lua

- Assimétricas
- Separação entre protótipo (estático) e co-rotina (dinâmico)
- `coroutine.create`, `coroutine.resume`, `coroutine.yield`

```
function f1 (v1)
  print("dentro", v1)
  io.read()
  local v2 = coroutine.yield(v1)
  print("dentro", v1+v2)
  io.read()
  return v1+v2
end
```

```
c1 = coroutine.create(f1)
io.read()
_,r1 = coroutine.resume(c1,10)
print("fora", r1)
io.read()
_,r2 = coroutine.resume(c1,20)
print("fora", r2)
io.read()
_,r3 = coroutine.resume(c1,30)
print("fora", r3)
```

# Co-rotinas

- Co-rotina = função + estado de execução
- Função
  - protótipo, estático
- Estado de Execução
  - pilha: locais
  - PC: contador do programa
  - dinâmico

# Exemplo: Iteradores

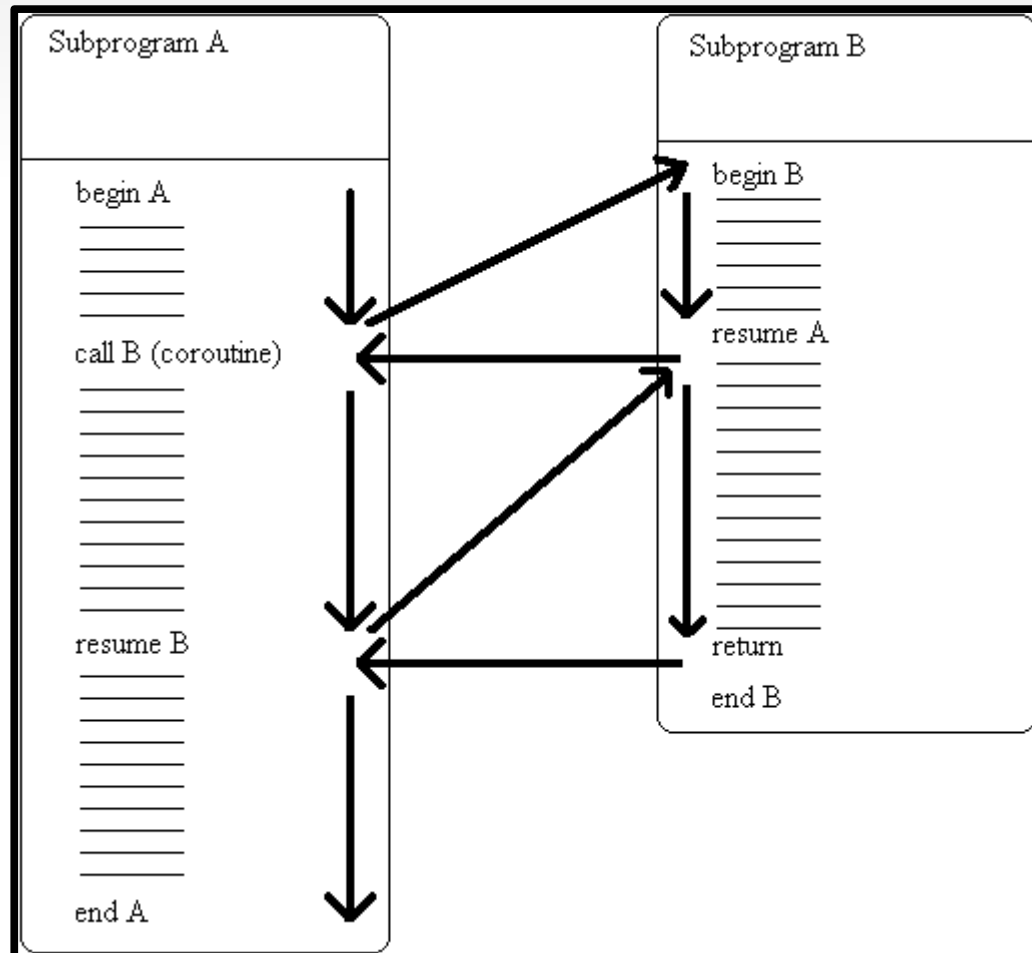
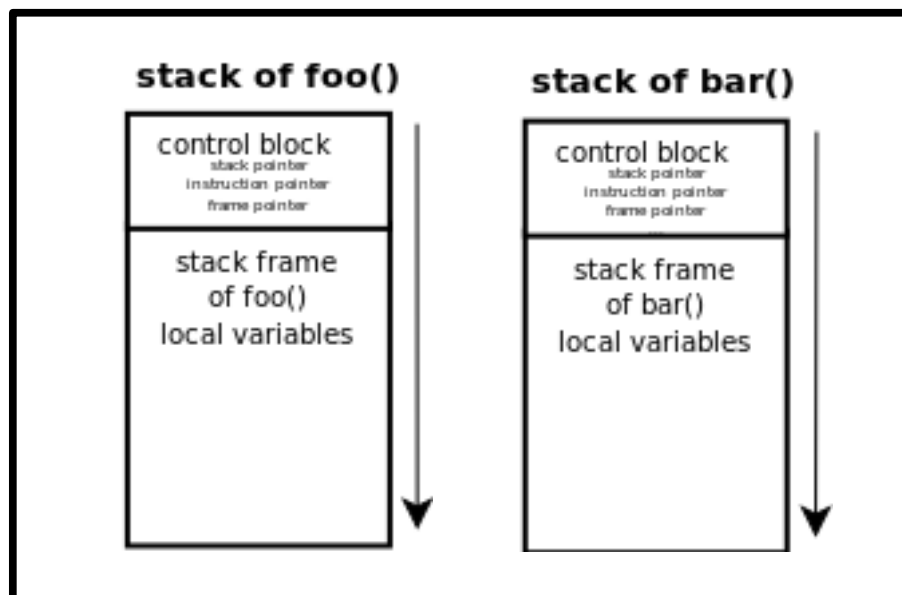
```
for i=1, 10 do  
    local v = i*i  
    print(i,v)  
end
```

```
for i,v in <f_iter> do  
    print(i,v)  
end
```

- `code/iterator-0[1-3].lua`
- Estado global e encapsulado

# Iteradores com Co-rotinas

- `code/iterator-0[4-5].lua`
- Contexto = PC, SP, pilha separada
  - estado implícito





# Co-rotinas

- Controle/Pilha como “cidadão de primeira classe”
- Iteradores, Multi-Tarefa cooperativa

Comparison with subroutines [ [edit](#) ]

"Subroutines are special cases of ... coroutines." -[Donald Knuth](#).<sup>[3]</sup>

# Exemplo: Joguinho

- Corrida entre dois jogadores
- `code/funs/coro-02.lua`
- **API:** `player1()`, `player2()`
- **Retorno:** `'move'` ou `'stand'`