

Compiladores II

Fabio Mascarenhas - 2014.2

<http://www.dcc.ufrj.br/~fabiom/comp2>

for genérico

- Vimos como usar o laço for genérico com as funções `ipairs` e `pairs`, mas não há nada de especial sobre essas funções
- A biblioteca padrão define outras funções que trabalham com o laço for genérico:

```
-- para cada linha em "foo.txt" faça...
for line in io.lines("foo.txt") do
  -- para cada palavra em line faça...
  for word in string.gmatch(line, "%w+") do
    print(word)
  end
  print("-----")
end
```

- Todas essas funções retornam iteradores

Iteradores

- Um iterador é uma função que, cada vez que é chamada, produz um ou mais itens que correspondem a um elemento de uma sequência
 - Cada índice e valor de um vetor
 - Cada chave e valor de uma tabela
 - Cada linha de um arquivo
 - Cada substring que casa com um padrão
- Quando acabaram os elementos o iterador retorna nil

for genérico e iteradores

- O for genérico recebe o iterador retornado pelas chamadas a `ipairs`, `pairs`, `io.lines`, e `string.gmatch`, e o chama repetidamente, associando os valores que ele retorna para suas variáveis de controle

```
> iter = function ()
>>     local x = math.random(4)
>>     if x == 4 then
>>         return nil
>>     else
>>         return x
>>     end
>> end
> for n in iter do print(n) end
1
3
1
```

Iteradores de fecho

- Um fecho é a maneira mais simples de definir um iterador útil:

```
function fromto(a, b)
  return function ()
    if a > b then
      return nil
    else
      a = a + 1
      return a - 1
    end
  end
end
```

iterador

- O fecho que fromto retorna é o iterador:

```
> for i in fromto(2, 5) do print(i) end
2
3
4
5
```

Quiz

- A função values retorna um iterador, o que ele produz?

```
function values(t)
  local i = 0
  return function ()
    i = i + 1
    return t[i]
  end
end
```

$i = 1$ $t[1]$
 $t[2]$
 $t[3]$

```
function ipairs(t)
  local i = 0
  return function ()
    i = i + 1
    return i, t[i]
  end
end
```

Programação Funcional

- Programação funcional é um estilo de programação em que programa-se usando valores imutáveis e funções de alta ordem
- Lua é essencialmente uma linguagem imperativa, então programação funcional não é o estilo usual de programar em Lua, mas podemos expresser alguns idiomas funcionais
- Linguagens funcionais geralmente usam listas ligadas para representar sequências de elementos, já que essa estrutura lida bem com imutabilidade
- Nós vamos usar iteradores ao invés de listas, o que dá características parecidas com o uso de listas em linguagens funcionais puras como Haskell

map e filter

- A função map transforma uma sequência, aplicando uma função a cada um de seus elementos:

```
> a = values{ 1, 2, 3, 4, 5 }
> b = map(function (x) return x * x end, a)
> print_iter(b)
{ 1, 4, 9, 16, 25 }
```

- filter transforma uma sequência, omitindo elementos que não passam por algum predicado:

```
> a = values{ 1, 2, 3, 4, 5 }
> b = filter(function (x) return x % 2 == 1 end, a)
> print_iter(b)
{ 1, 3, 5 }
>
```


Folds

- Um fold é uma redução de uma sequência usando uma operação binária e uma semente

- Um fold à esquerda começa aplicando a operação à semente e ao primeiro elemento, depois aplica a operação ao resultado e o segundo elemento, assim por diante $((((s \oplus x_1) \oplus x_2) \oplus x_3) \dots \oplus x_m)$

- Um fold à direita começa aplicando a operação ao último elemento e à semente, depois ao penúltimo elemento e ao resultado, e assim por diante

$$(x_1 \oplus \dots \oplus (x_2 \oplus (x_m \oplus s)))$$

- Folds à esquerda são mais eficientes de calcular (por quê?)

espaço constante

Dojo

- Implementar as funções map, filter, foldl e foldr, dada a especificação nos slides anteriores

Metatabelas

- Uma *metatabela* modifica o comportamento de outra tabela; usando uma metatabela com os campos apropriados nós podemos:
 - Usar operadores aritméticos, relacionais e de concatenação
↳ subscrita estilo C++ / C#
 - Alterar o comportamento dos operadores ==, ~= e #
 - Alterar o comportamento das funções embutidas tostring, pairs e ipairs
 - Prover valores para campos inexistentes, e interceptar a criação de novos campos
 - Chamar uma tabela como uma função

t (...)

Escopo das metatabelas

- Cada tabela pode ter sua própria metatabela, que vai mudar o comportamento apenas daquela tabela
- Mas várias tabelas podem compartilhar uma única metatabela, de modo que todas tenham comportamento similar "columns" "PDTs"
- A função embutida `setmetatable` muda a metatabela de uma tabela, e retorna essa tabela (não a metatabela!)
- A função embutida `getmetatable` retorna a metatabela de uma tabela, ou nil se ela não tiver uma
- Não é recomendado modificar uma metatabela depois de associá-la a uma tabela, pois isso tem impacto no desempenho

Metamétodos

- Especificamos as operações que a metatabela vai modificar atribuindo a *metamétodos*
- Um metamétodo é uma função (ou tabela) associada a um campo com um nome pré-definido
- Existem 19 metamétodos: `__add`, `__sub`, `__mul`, `__div`, `__mod`, `__pow`, `__unm`, `__concat`, `__len`, `__eq`, `__lt`, `__le`, `__index`, `__newindex`, `__call`, `__tostring`, `__ipairs`, `__pairs`, `__gc`
- Quase todos devem ser funções, exceto por `__index` e `__newindex`, que podem ser tabelas; usar uma tabela para `__index` é a base para a programação OO com herança simples em Lua