Compiladores II

Fabio Mascarenhas - 2014.2

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

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
  - 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

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 "classes" "ADTs"

- 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

Números complexos

- Como exemplo, vamos usar metamétodos para melhorar o módulo de números complexos visto na última aula com várias operações:
  - Adição a reais e outros números complexos usando + (a mesma técnica funciona para os outros operadores aritméticos)
  - Comparação estrutural para igualdade
  - Módulo de um número complexo usando #
  - *Pretty-printing* com `tostring`

Compartilhando uma metatabela

- Primeiro criamos uma tabela privada ao módulo e a atribuímos como metatabela de cada número complexo que criamos com `new`:

```
local mt = {}

local function new(r, i)
  return setmetatable({ real = r or 0, im = i or 0 }, mt)
end
```

- Essa metatabela também nos dá um bom teste para ver se algum valor qualquer é um número complexo ou não:

```
local function is_complex(v)
  return getmetatable(v) == mt
end
```

Sobrecarregando + com __add

- A função add já soma dois complexos; se nós a atribuirmos ao campo __add da nossa metatabela, + começará a funcionar para pares de números complexos:

```
local function add(c1, c2)
  return new(c1.real + c2.real, c1.im + c2.im)
end

mt.__add = add
```

```
> c1 = complex.new(2, 3)
> c2 = complex.new(1, 5)
> print(complex.tostring(c1 + c2))
3+8i
```

- Vamos ver o que acontece se tentarmos somar um real a um complexo:

```
> c3 = c1 + 5
.\complex.lua:20: attempt to index local 'c2' (a number value)
stack traceback:
        .\complex.lua:20: in function '__add'
        stdin:1: in main chunk
        [C]: in ?
```

Resolução de aritmética

- O que está acontecendo? Lua está chamando o metamétodo `__add` do número complexo! Se o operando esquerdo da soma tiver um metamétodo `__add` então Lua irá sempre chamá-lo. Podemos tirar proveito disso:

```
local function add(c1, c2)
  if not is_complex(c2) then
    return new(c1.real + c2, c1.im)
  end
  return new(c1.real + c2.real, c1.im + c2.im)
end
```

- A soma de um real a um complexo agora funciona:

```
> c1 = complex.new(2, 3)
> c3 = c1 + 5
> print(complex.tostring(c3))
7+3i
```

Resolução de aritmética (2)

- E quanto a somar um complexo a um real?

```
> c3 = 5 + c1
.\complex.lua:20: attempt to index local 'c1' (a number value)
stack traceback:
        .\complex.lua:20: in function '__add'
        stdin:1: in main chunk
        [C]: in ?
```

- Se o operando esquerdo não tem um metamétodo mas o direito tem, Lua vai chamar o metamétodo do operando direito, o que nos dá a forma final de add:

```
local function add(c1, c2)
  if not is_complex(c1) then
    return new(c2.real + c1, c2.im)
  end
  if not is_complex(c2) then
    return new(c1.real + c2, c1.im)
  end
  return new(c1.real + c2.real, c1.im + c2.im)
end
```

```
> c3 = 5 + c1
> print(complex.tostring(c3))
7+3i
```

Igualdade

- O metamétodo `__eq` controla tanto == quanto ~=

- A resolução é diferente da aritmética, já que Lua vai chamar o metamétodo apenas se ambos os operandos têm a mesma metatabela. Isso nos dá a seguinte implementação para igualdade de complexos:

```
local function eq(c1, c2)
  return (c1.real == c2.real) and (c1.im == c2.im)
end

mt.__eq = eq
```

```
> c1 = complex.new(1, 2)
> c2 = complex.new(2, 3)
> c3 = complex.new(3, 5)
> print(c1 + c2 == c3)
true
> print(c1 ~= c2)
true
```

- Uma limitação é que a comparação de um complexo com um real sempre vai ser falsa, mesmo que a parte imaginária seja zero

Sobrecarregando # e `tostring`

- Tanto o metamétodo __len quanto __tostring trabalham da mesma forma: eles recebem seu operando e devem retornar seu resultado, o que dá uma implementação simples para nós:

```
local function modulus(c)
  return math.sqrt(c.real * c.real + c.im * c.im)
end

mt.__len = modulus

local function tos(c)
  return tostring(c.real) .. "+" .. tostring(c.im) .. "i"
end

mt.__tostring = tos
```

```
> c1 = complex.new(3, 4)
> print(#c1)
5
> print(tostring(c1))
3+4i
> print(c1)
3+4i
```

- `print` usa `tostring`

Operadores relacionais

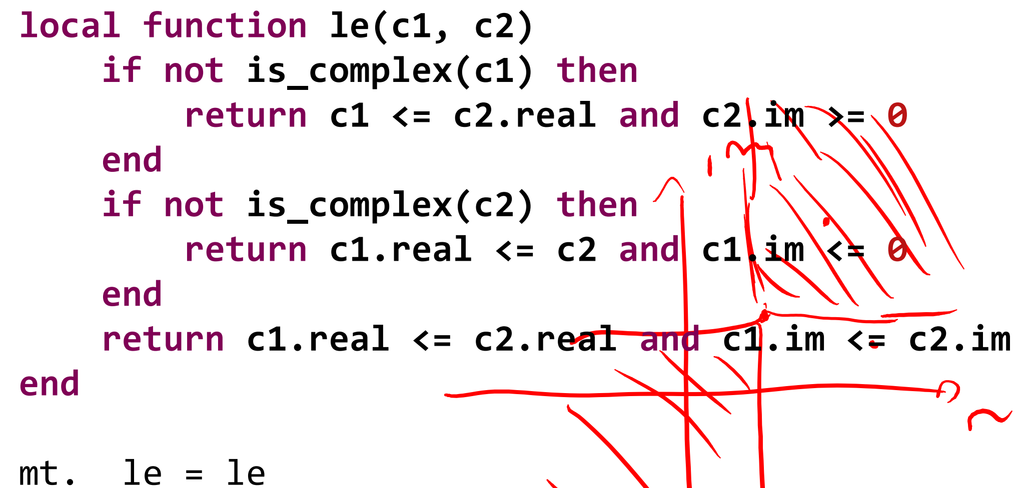
- O metamétodo para < (`__lt`) funciona do mesmo modo que os metamétodos aritméticos; para >, Lua usa `__lt` invertendo a ordem dos operandos
- O metamétodo para <= (`__le`) é similar, mas se ele não estiver definido e `__lt` estiver Lua usa `__lt`, invertendo a ordem e negando o resultado
- Para que usar dois metamétodos? Para *ordens parciais*:

```
local function lt(c1, c2)
    if not is_complex(c1) then
        return c1 < c2.real and c2.im > 0
    end
    if not is_complex(c2) then
        return c1.real < c2 and c1.im < 0
    end
    return c1.real < c2.real and c1.im < c2.im
end

mt.__lt = lt
```

```
local function le(c1, c2)
    if not is_complex(c1) then
        return c1 <= c2.real and c2.im >= 0
    end
    if not is_complex(c2) then
        return c1.real <= c2 and c1.im <= 0
    end
    return c1.real <= c2.real and c1.im <= c2.im
end

mt.__le = le
```



`__index` e `__newindex`

- Se uma tabela não tem uma chave mas sua metatabela tem um metamétodo `__index` então Lua vai chamar este metamétodo, passando a tabela e a chave sendo procurada, e o que o metamétodo retornar vai ser o resultado da indexação

- Em uma atribuição a uma chave inexistente, se a metatabela tem um metamétodo `__newindex` então Lua vai chamá-lo, passando a tabela, a chave e o valor sendo atribuído

- Um uso comum destes metamétodos é associá-los a uma tabela vazia para criar uma *tabela proxy,* que é mantida vazia para que todas as operações de indexação sejam interceptadas

- Tanto `__index` quanto `__newindex` podem ser tabelas ao invés de funções, e nesse caso Lua refaz a operação de indexação no metamétodo

Um proxy para contagem de acesso

```lua
local mt = {}

function mt.__index(t, k)
  t.__READS = t.__READS + 1
  return t.__TABLE[k]
end

function mt.__newindex(t, k, v)
  t.__WRITES = t.__WRITES + 1
  t.__TABLE[k] = v
end

local function track(t)
  local proxy = { __TABLE = t, __READS = 0, __WRITES = 0}
  return setmetatable(proxy, mt)
end

return { track = track }
```

```
> proxy = require "proxy"
> t = proxy.track({})
> t.foo = 5
> print(t.foo)
5
> t.foo = 2
> print(t.__READS, t.__WRITES)
1       2
```

Quiz

- Podemos contornar a limitação de `__eq de modo a ter` `complex.new(2,0) == 2` fazendo `complex.new` retornar um real se a parte imaginária for 0. Alguma operação deixará de funcionar com essa mudança? Qual(is)?

is_complex

Métodos e :

- Na maioria das linguagens OO, um *método* tem um *destinatário* implícito, normalmente chamado de *this* ou *self,* em adição a seus outros parâmetros

- Em Lua, um método é apenas uma função que recebe o destinatário como primeiro parâmetro, e o programador pode chamá-la do jeito que preferir

- Indexar um objeto Lua com o nome do método retorna sua função, e então podemos chamar o método:

```
> obj.method(obj, <outros argumentos>)
```

- Para evitar a repetição do destinatário usamos o operador *dois pontos*:

```
> obj:method(<outros argumentos>)
```

- O operador acrescenta o destinatário como primeiro argumento da chamada; o destinatário pode ser qualquer expressão, que é avaliada só uma vez, mas o nome do método tem que ser um identificador válido

Declarando métodos

- Podemos usar dois pontos para declarar um método, com o efeito de atribuir uma função com um parâmetro `self` extra:

```
function obj:method(<other arguments>)
  <código do método>
end
```

≡

```
function obj.method(self, <other arguments>)
  <código do método>
end
```

- Definindo um objeto simples:

```
local square = { x = 10, y = 20, side = 25 }

function square:move(dx, dy)
  self.x = self.x + dx
  self.y = self.y + dy
end

function square:area()
  return self.side * self.side
end

return square
```

```
> print(square:area())
625
> square:move(10, -5)
> print(square.x, square.y)
20      15
```

Classes

- Os métodos de `square` funcionam com qualquer tabela que tenha os campos `x`, `y` e `side` como destinatária:

```
> square2 = { x = 30, y = 5, side = 10 }
> print(square.area(square2))
100
> square.move(square2, 10, 10)
> print(square2.x, square2.y)
40      15
```

- Podemos definir esses métodos dentro de uma *classe* `Square`, que funciona como um *protótipo* para objetos como `square` e `square2`, junto com um método `new` para criar novas *instâncias*

- Essas instâncias têm valores para seus campos `x`, `y` e `side`, e uma metatabela com metamétodo `__index` apontando para `Square`

Square

- Uma maneira de escrever `Square`, como um pacote:

```
local Square = {}
Square.__index = Square

function Square:new(x, y, side)
  return setmetatable({ x = x, y = y, side = side }, self)
end

function Square:move(dx, dy)
  self.x = self.x + dx
  self.y = self.y + dy
end

function Square:area()
  return self.side * self.side
end

return Square
```

```
> s1 = Square:new(10, 5, 10)
> s2 = Square:new(20, 10, 25)
> print(s1:area(), s2:area())
100     625
> s1:move(5, 10)
> print(s1.x, s1.y)
15      15
```

→ clone

Campos com valores default

- Outros campos que acrescentarmos no construtor de `Square` serão valores default para os campos nas instâncias:

```
local Square = { color = "blue" }
```

- Lendo o campo pegamos o valor default:

```
> s1 = Square:new(10, 5, 10)
> print(s1.color)
blue
```

- Atribuindo ao campo mudamos o valor dele para aquela instância, sem afetar as outras:

```
> s1.color = "red"
> print(s1.color)
red
> s2 = Square:new(20, 10, 25)
> print(s2.color)
blue
```

Circle

- Vamos criar outra classe, `Circle`:

```
local Circle = {}
Circle.__index = Circle

function Circle:new(x, y, radius)
  return setmetatable({ x = x, y = y, radius = radius }, self)
end

function Circle:move(dx, dy)
  self.x = self.x + dx
  self.y = self.y + dy
end

function Circle:area()
  return math.pi * self.radius * self.radius
end

return Circle
```

- O método `move` é idêntico ao de `Square`!

Shape

- Podemos fatorar as partes comuns entre `Square` e `Circle` em uma classe `Shape`:

```
local Shape = {}
Shape.__index = Shape

function Shape:new(x, y)
  return setmetatable({ x = x, y = y }, self)
end

function Shape:move(dx, dy)
  self.x = self.x + dx
  self.y = self.y + dy
end

return Shape
```

- A metatabela de uma instância é sua classe; a metatabela de uma classe será sua *superclasse*

Point extends Shape

- Pontos são figuras com coordenadas x e y e área 0:

```lua
local Shape = require "shape"
local Point = setmetatable({}, Shape)
Point.__index = Point

function Point:area()
  return 0
end

return Point
```

```
> p = Point:new(10, 20)
> print(p:area())
0
> p:move(-5, 10)
> print(p.x, p.y)
5       30
```

- A chamada a `setmetatable` quando definimos a nova classe faz ela herdar os métodos de `Shape`, incluindo seu “construtor” `new`

`Circle` extends `Shape`

- Precisamos redefinir o construtor em `Circle`, mas podemos usar o construtor de `Shape` para fazer parte do trabalho:

```
local Shape = require "shape"
local Circle = setmetatable({}, Shape)
Circle.__index = Circle

function Circle:new(x, y, radius)
  local shape = Shape.new(self, x, y)
  shape.radius = radius
  return shape
end

function Circle:area()
  return math.pi * self.radius * self.radius
end

return Circle
```

Usamos o mesmo truque para chamar o método “super” em outros métodos redefinidos

```
> c = Circle:new(10, 20, 5)
> c:move(5, -5)
> print(c.x, c.y)
15      15
> print(c:area())
78.539816339745
```

Outros modelos de objeto

- Esse é apenas um jeito de implementar objetos em Lua
- Ele tem a desvantagem de misturar "métodos de classe" (`new`) e "métodos de instância" (`move`, `area`) no mesmo espaço de nomes
- Outros metamétodos não são herdados; por exemplo, se quisermos conectar `__tostring` com um método `tostring` que pode ser redefinido (como `toString` de Java) precisamos explicitamente fazer `*Class*.__tostring = *Class*.tostring` em cada classe
- Mas a vantagem é que ele é simples! Modelos mais sofisticados existem como bibliotecas, e todos eles suportam o uso do operador `:` para chamadas de métodos

Dojo

- Strings também são objetos em Lua, e seus métodos são as funções do módulo `string`, logo podemos usar o operador `:` com elas:

```
> =("ola mundo"):byte(2, 5)
108     97      32      109
```

- Defina uma classe `Seq` para representar sequências que implementa parte do protocolo das strings: um método `len` para dar o tamanho da sequência, um método `sub` para retornar uma subsequência, um método `byte` para retornar elementos da sequência, mais metamétodos `__concat` e `__tostring` para concatenar sequências e pretty-printing do seu conteúdo; o construtor de uma sequência recebe um vetor com seus elementos

- Modifique o scanner para operar em strings ou sequências de bytes como entrada ao invés de apenas strings