Compiladores II

Fabio Mascarenhas - 2014.2

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

Dojo

- O objetivo é construir um tokenizador para um subconjunto de Lua, a função token deve receber uma string, e retornar o primeiro token da string (ignorando espaços) e o que sobrou depois de consumir esse token; um token é um registro contendo o tipo do token e o lexema
- Palavras reservadas: apenas function, end, while, local, true, and, false, else, if, elseif, not, nil, or, return, then, do
- Numerais: apenas inteiros e decimais sem notação científica
- Strings: apenas aspas duplas
- Operadores: apenas +, -, *, /, ==, ~=, <, =, (,), {, }, ., ,, ..
- Identificadores

Erros

- Funções Lua têm duas maneiras de sinalizar erros: retornar `nil` e uma mensagem de erro, ou lançar um erro

- A primeira maneira é para quando os erros são esperados, quando estamos abrindo um arquivo, por exemplo, já que sempre há o risco do arquivo não existir:

```
> print(io.open("notafile.txt"))
nil     notafile.txt: No such file or directory 2
```

- A segunda maneira é para problemas excepcionais, quando há erros na entrada, ou bugs no código:

```
> print(math.sin("foo"))
stdin:1: bad argument #1 to 'sin' (number expected, got string)
stack traceback:
        [C]: in function 'sin'
        stdin:1: in main chunk
        [C]: in ?
```

De mensagens de erro para erros

- A função embutida `assert` transforma erros do primeiro tipo em erros do segundo:

```
> print(assert(io.open("foo.txt")))
file (000007FF650BE2D0)
> print(assert(io.open("notafile.txt")))
stdin:1: notafile.txt: No such file or directory
stack traceback:
        [C]: in function 'assert'
        stdin:1: in main chunk
        [C]: in ?
```

- A função embutida `error` recebe uma mensagem e lança um erro:

```
> error("raising an error")
stdin:1: raising an error
stack traceback:
        [C]: in function 'error'
        stdin:1: in main chunk
        [C]: in ?
```

Divisão inteira, com e sem erros

- As duas implementações da função `idiv` abaixo mostram as duas maneiras de reporter erros:

```lua
function idiv1(a, b)
  if b == 0 then
    return nil, "division by zero"
  else
    return math.floor(a/b)
  end
end


function idiv2(a, b)
  if b == 0 then
    error("division by zero")
  else
    return math.floor(a/b)
  end
end
```

```
> print(idiv1(2,0))
nil     division by zero
> print(idiv2(2,0))
stdin:3: division by zero
stack traceback:
        [C]: in function 'error'
        stdin:3: in function 'idiv2'
        stdin:1: in main chunk
        [C]: in ?
```

Capturando erros

- Lançar um erro aborta a execução por default; se estivermos no REPL voltamos ao prompt

- Podemos capturar e tratar erros usando a função embutida `pcall`, que recebe uma função para chamar e seus argumentos, retornando:

  protected call

  - `true` seguido dos resultados da função, se não houve erros

  - `false` seguido da mensagem de erro, se houve um erro

```
> print(pcall(idiv2, 5, 2))
true    2
> print(pcall(idiv2, 5, 0))
false   division by zero
```

Módulos

- Até agora estivemos trabalhando no REPL e no context de um único script
- Também estivemos usando funções embutidas como `ipairs` e `table.concat`
- Mas a maior parte das aplicações fica melhor distribuída em vários scripts, e não usam apenas as funções embutidas
- Módulos resolvem tanto o problema do compartilhamento quanto do reuso de código; um modulo é um grupo reusável de funções e estruturas de dados relacionadas

Módulos são tabelas

- Um módulo Lua é um pedaço de código que cria e retorna uma tabela; essa tabela tem todas as funções e estruturas de dados que o módulo exporta

- A biblioteca padrão define vários módulos embutidos: table, io, string, math, os, coroutine, package e debug

- Uma aplicação carrega um módulo com a função embutida `require`; ela recebe o nome de um *pacote*, e retorna um módulo

- A aplicação deve atribuir o módulo retornado por `require` a uma variável, já que `require` por si só não faz nenhuma atribuição

Um pacote simples

- Um pacote é um script Lua que cria e retorna um módulo; como exemplo, vamos criar um módulo simples para números complexos em um pacote "complex.lua" no mesmo path onde estamos rodando nosso REPL:

```lua
local M = {}

function M.new(r, i)
  return { real = r or 0, im = i or 0 }
end

M.i = M.new(0, 1)

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

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

return M
```

Outro estilo

- Podemos definir o mesmo módulo em um estilo diferente:

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

local i = new(0, 1)

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

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

return { new = new, i = i, add = add, tostring = tos }
```

- Esse estilo tem melhor desempenho, mas mais duplicação; é questão de gosto

Carregando o pacote `complex`

- Podemos carregar nosso novo pacote no REPL:

```
> complex = require "complex"
> print(complex)
table: 0000000000439820
```

- Se chamarmos `require` novamente recebemos nosso módulo cacheado:

```
> print(require "complex")
table: 0000000000439820
```

- Para forçar a recarga do pacote limpamos o módulo do cache:

```
> package.loaded.complex = nil
> complex = require "complex"
> print(complex)
table: 000000000042F8F0
```

Usando o módulo

- Uma vez que temos o módulo em uma variável, podemos usar qualquer coisa que ele exporta:

```
> c1 = complex.new(1, 2)
> print(complex.tostring(c1))
1+2i
> c2 = complex.add(c1, complex.new(10,2))
> print(complex.tostring(c2))
11+4i
> c3 = complex.add(c2, complex.i)
> print(complex.tostring(c3))
11+5i
```

- Um módulo é uma tabela, então poderíamos atribuir a seus campos e substituir suas funções (*monkey-patching*), mas isso não é um estilo de programação recomendado em Lua

Procurando pacotes

- Onde `require` procura o pacote `complex.lua`? Ele usa um caminho de busca na variável `package.path`:

```
> print(package.path)
/usr/local/share/lua/5.2/?.lua;/usr/local/share/lua/5.2/?/init.lua;/usr/local/li
b/lua/5.2/?.lua;/usr/local/lib/lua/5.2/?/init.lua;./?.lua
```

- O caminho de busca depende do sistema, e vem por default da variável de ambiente `LUA_PATH_5_2`, se definida, ou `LUA_PATH`, ou um valor pré-compilado no interpretador

- O caminho de busca é uma lista de modelos separados por ponto-e-vírgula; `require` tenta cada modelo em sequência, substituindo `?` pelo nome do pacote

Procurando `complex`

- Para o caminho de busca no slide anterior, `require` vai tentar os seguintes caminhos:

```
/usr/local/share/lua/5.2/complex.lua
/usr/local/share/lua/5.2/complex/init.lua
/usr/local/lib/lua/5.2/complex.lua
/usr/local/lib/lua/5.2/complex/init.lua
./complex.lua
```

- Como `complex.lua` está no caminho corrente o último é bem sucedido

- Podemos usar a função `package.searchpath` para ver qual pacote será carregado para determinado nome e caminho de busca:

```
> print(package.searchpath("complex", package.path))
.\complex.lua
```

Conflitos

- Suponha que temos dois pacotes complex.lua em nosso sistema, mas eles significam coisas diferentes, ou implementações diferentes da mesma coisa, e queremos ter as duas

- Podemos por cada pacote em sua própria pasta; por exemplo, o primeiro em `adts/complex.lua`, e o segundo em `numlua/complex.lua`

- Podemos fazer `require "adts.complex"` para carregar o primeiro, e `require "numlua.complex"` para o segundo

- Lua substitui os pontos pelo separador de caminho do sistema antes de começar a procura pelo pacote

```
> print(package.searchpath("adts.complex", package.path))
nil
        no file '/usr/local/share/lua/5.2/adts/complex.lua'
        no file '/usr/local/share/lua/5.2/adts/complex/init.lua'
        no file '/usr/local/lib/lua/5.2/adts/complex.lua'
        no file '/usr/local/lib/lua/5.2/adts/complex/init.lua'
        no file './adts/complex.lua'
```

Iterando sobre ...

- Podemos usar ... dentro de um constructor de tabela, e assim iterar sobre os argumentos extras de uma função variádica:

```
function add(...)
  local sum = 0
  for _, n in ipairs({ ... }) do
    sum = sum + n
  end
  return sum
end
```

- Se algum dos argumentos extras for nil então { ... } não será um vetor. A função table.pack junta todos os seus argumentos, inclusive nils, em uma tabela, e escreve o número de argumentos no campo “n” :

```
> t = table.pack(1, nil, 3)
> for i = 1, t.n do print(t[i]) end
1
nil
3
```

table.unpack

require("foo")

- A inversa de table.pack é a função table.unpack:

```
> print(table.unpack{ 1, 2, 3, 4 })
1       2       3       4
```

- Da maneira acima, table.unpack só funciona com vetores sem buracos

- Para “vetores” com buracos, table.unpack aceita mais dois argumentos, que dão os índices inicial e final do interval retornado por table.unpack:

```
> a = { [2] = 5, [5] = 0 }
> print(table.unpack(a, 1, 5))
nil     5       nil     nil     0
```

1 2 3 4 5

“Argumentos” nomeados

- Pode-se simular uma função que recebe parâmetros nomeados passando um registro:

```
function rename(args)
  return os.rename(args.old, args.new)
end
```

- Lua tem um pouco de suporte sintático para esse uso; se o único argumento de uma função for um construtor, os parênteses podem ser omitidos:

```
rename{ new = "perm.lua", old = "temp.lua" }
```

Escopo léxico

- Qualquer variável local visível no ponto em que uma função é definida também é visível dentro da função, podendo inclusive ser modificada:

```
function derivative(f, dx)
  dx = dx or 1e-4
  return function (x)
           -- tanto f quanto dx visíveis!
           return (f(x + dx) - f(x)) / dx
         end
end
```

- A função derivative recebe uma função e retorna outra função, e é um exemplo de função de alta ordem:

```
> df = derivative(function (x) return x * x * x end)
> print(df(5))
75.001500009932
```

Fechos

- Dizemos que uma função se fecha sobre as variáveis locais que ela usa, então chamamos essas funções de fechos (closures)

- Um fecho pode tanto ler quanto escrever as variáveis locais que ela se fechou sobre:

```
function counter()
  local n = 0
  return function ()
           n = n + 1
           return n
         end
end
```

```
> c1 = counter()
> c2 = counter()
> print(c1())
1
> print(c1())
2
> print(c2())
1
```

- Cada chamada a counter() cria um novo fecho

- Cada fecho se fecha sobre uma instância diferente de n

Fechos e compartilhamento

- Fechos se fecham sobre as próprias variáveis, e não sobre cópias delas, então dois fechos podem compartilhar uma variável:

```
function counter()
  local n = 0
  return function (x)
           n = n + (x or 1)
           return n
         end,
         function (x)
           n = n - (x or 1)
           return n
         end
end
```

- counter() agora retorna dois fechos com o mesmo n

```
> inc, dec = counter()
> print(inc(5))
5
> print(dec(2))
3
> print(inc())
4
```

Callbacks

- Fechos são um bom mecanismo para callbacks; por exemplo, table.sort recebe como parâmetro opcional um callback para comparar os elementos do vetor:

```
> a = { "Python", "Lua", "C", "JavaScript", "Java", "Lisp" }
> table.sort(a, function (a, b) return a > b end)
> print_array(a)
{ Python, Lua, Lisp, JavaScript, Java, C }
```