Programming in Lua – More about Functions

Fabio Mascarenhas

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

Iterating over ...

- You can collect and then iterate over all the extra arguments to a variadic function using `...` inside a table constructor:

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

- If any of the extra arguments is nil then { ... } will not be an array, so you need to use the `table.pack` function to collect the arguments in a table with the field “n” set to the number of arguments:

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

table.unpack

- The flip side of `table.pack` is the function `table.unpack`, to return all elements of an array in order:

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

- Using table.unpack this way is only guaranteed to work for proper arrays (without holes)

- You can pass two more arguments to table.unpack, for the starting and ending indices, and unpack will return all elements in the interval regardless of holes

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

“Named” arguments

- You can simulate a function that takes named arguments with a function that takes a record:

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

- If you are calling a function and passing a single table constructor, you can omit the parentheses:

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

- You can put spaces between the function and `{`, but it is good style to omit the spaces

Lexical scoping

- Any local variable visible in the point where a function is defined is also visible inside the function (as long as it is not shadowed by parameters or local variables inside the function):

```
function derivative(f, dx)
  dx = dx or 1e-4
  return function (x)
           -- both f and dx visible here!
           return (f(x + dx) - f(x)) / dx
         end
end
```

- The derivate function takes a function and returns another function, and is an example of a *higher-order function*:

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

Closures

- We say that a function *closes over* the local variables from its surroundings that the function uses, so we call these functions *closures*

- A closure can not only read but also assign to the local variables it closes over:

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

- Each call to `counter()` creates a new closure

- Each closure closes over a different instance of n

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

Closures and sharing

- Closures do not close over copies of local variables, but over the variables themselves, so two closures can *share* a single variable:

```
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() now returns two closures that share the same n

- And the only way to access n is through the closures!

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

Callbacks

- Lua closures are a nice and lightweight mechanism for *callbacks*; for example, `table.sort` takes as optional second argument a callback that must tell whether element *a* comes before element *b* in the sorted array:

```
> 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 }
```

- Callbacks are also very common in GUI code, as a way of responding to user events, and for asynchronous code

Functional Programming

- *Functional programming* is a programming style where we program using immutable values and higher-order functions
- Lua is in essence an *imperative* language, so functional programming is not the usual style, but we can easily do functional programming using Lua
- Functional languages commonly use linked lists to represent sequences of elements, as they play well with immutability
- We will use Lua arrays, which will have different performance characteristics, but will be more compact

map and filter

- The map function iterates over a sequence, applying a function to each element and collecting the results in another sequence:

```
function map(f, l)
  local nl = {}
  for i, x in ipairs(l) do
    nl[i] = f(x)
  end
  return nl
end
```

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

- Filter iterates over a sequence, collecting the elements that pass a predicate:

```
function filter(p, l)
  local nl = {}
  for _, x in ipairs(l) do
    if p(x) then
      nl[#nl+1] = x
    end
  end
  return nl
end
```

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

Folds

- A *fold* is a reduction of a sequence using a binary operation and a *seed*
- A *left fold* starts by applying the operation to the seed and the first element, then applying the operation to the result and the second element, and so on
- A *right fold* starts by applying the operation to the last element and the seed, then applying the operation to the second-to-last element and the result, and so on

```
function foldl(op, z, l)
  for _, x in ipairs(l) do
    z = op(z, x)
  end
  return z
end
```

```
function foldr(op, z, l)
  for i = #l, 1, -1 do
    z = op(l[i], z)
  end
  return z
end
```

Currying

- A *curried* function is a function that, instead of taking all of its parameters at once, takes a proper prefix of them and then returns a (possibly also curried) function that takes the rest of the parameters; for example, the following is a curried version of map:

```
function map(f)
  return function(l)
           local nl = {}
           for i, x in ipairs(l) do
             nl[i] = f(x)
           end
           return nl
         end
end
```

- Currying makes it easy to do *partial evaluation* of functions

```
> square = map(function (x) return x * x end)
> print_array(square{ 1, 5, 9 })
{ 1, 25, 81 }
```

Quiz

- What is wrong with the function `named` below, that turns a function with positional arguments into a function with named arguments? How to fix it?

```
function named(f, names)
  return function (args)
           local l = map(function (name) return args[name] end, names)
           f(table.unpack(l))
         end
end

rename = named(os.rename, { “old”, “new” })
rename{ old = “old.txt”, new = “new.txt” }
```