中国科学技术大学 University of Science and Technology of China

Lua

Foundation of PLs

Yu Zhang

Acknowledgement: Stanford CS242: Programming Languages, http://cs242.stanford.edu/

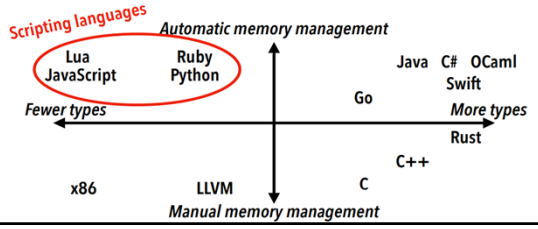
Course web site: http://staff.ustc.edu.cn/~yuzhang/fopl

Outline

- Lua Introduction
- Lua History
- Lua Features

What is a scripting language?

- Common features of a scripting language
  - Dynamically typed
  - Garbage collected
  - Rich standard library
  - Reflection/metaprogramming



Scripting langs: so productive

- Key idea: encode program information as you go
  - e.g. type information, data lifetime
  - No one likes commitment!
- Easy to use certain idioms hard to express in static types
  - Interfaces/duck typing
  - Polymorphism
  - Heterogeneous data structures
  - Extensible classes

Script. langs: semi-specialized

- Bash-like scripting languages
  - e.g. Python, Perl
  - For file manipulation,data crunching, command line parsing
- Web scripting languages
  - e.g. JavaScript, PHP
  - Specialized constructs for dealing with webpages or HTTP requests
- Embedded scripting languages
  - Mostly just Lua
  - Lightweight, easy to build, simple semantics for games, config files

Why Lua?

- Simplest, cleanest scripting language still in use
- "Correct" scoping
- No class system (but can build our own!)
- Easy to learn in a day
- Born in 1993 at PUC-Rio, Brazil
- https://www.lua.org/pil/
- Data structure: *tables* (associative arrays)
- Coroutines, extensible semantics, embedding

The Evolution of Lua, HOPL III, Jun 9-10, 2007.

Lua: Overview

- Lua function

```
-- Recursive impl.
function fact(n)
  if n == 0 then
    return 1
  else
    return n * fact(n-1)
  end
end
```

```
-- Iterative impl.
function fact(n)
  local a = 1
  for i = 1, n do
    a = a * i
  end
  return a
end
```

- C

```
-- Recursive impl.
int fact(int n) {
  if (n == 0)
    return 1;
  else
    return n * fact(n-1);
}
```

```
-- Iterative impl.
int fact(int n){
  int a = 1, i;
  for (i = 1; i<n; i++)
    a = a * i;
  return a;
}
```


Lua Implementation

- https://www.lua.org/download.html
  - Lua 5.3: Jan 12, 2015, Lua 5.3.5: Jul 10, 2018
  - Smallish, e.g. Lua 5.1 17000 lines of C
  - Portable
  - Embeddable
    - can call Lua from C and C from Lua
  - Clean code
    - Good for your “code reading club”
  - Efficiency
    - Fast for an interpreted scripting language: lang. simplicity helps
    - Presently has a register based VM, pre-compilation supported

Lua vs. Modula Syntax

- Designed for productive use
  - “Syntactically, Lua is reminiscent of Modula and uses familiar keywords.” [HOPL]

```
-- Lua
function fact(n)

  local a = 1


  for i = 1, n do
    a = a * i
  end
  return a
end
```

```
-- Modula-2
PROCEDURE Fact(n: CARDINAL):
      CARDINAL;
  VAR a: CARDINAL;
BEGIN
  a := 1
  FOR i := 1 TO n DO
    a := a * i;
  END;
  RETURN a;
END Fact;
```


Lua: similarities with Scheme

“The influence of Scheme has gradually increased during Lua’s evolution.”

- Simiarities
  - Dynamic typing, first-class values, anonymous functions, closures, ...
    - would have wanted first-class continuations
    - function foo() ... end is syntactic sugar for foo = function () ... end
  - Scheme has lists as its data structuring mechanism, while Lua has tables.
  - No particular object or class model forced onto the programmer—choose or implement one yourself.

Lua History

- Prehistory
  - Born in 1993 inside Tecgraf
    (Comp. Graphics Tech. Group of PUC-Rio in Brazil)
  - Lua creators: Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes
  - **Lua Ancestors:** “These languages, called DEL and SOL, were the ancestors of Lua.”
    - DEL and SOL were domain-specific languages (DSLs) for Tecgraf-developed interactive graphical programs
  - DEL as in "data-entry language“
    - for describing **data-entry tasks**: named and typed fields, data validation rules, how to input and output data

SOL

- SOL as in "Simple Object Language"
  - a DSL for a configurable report generator for lithology profiles
- SOL interpreter
  - read a report description, and syntax and type check specified objects and attributes
- syntax influenced by BibTeX

```
type @track{ x:number, y:number=23, id=0 }
type @line{ t:@track=@track{x=8}, z:number* }
T = @track{ y=9, x=10, id="1992-34" }
L = @line{ t=@track{x=T.y, y=T.x}, z=[2,3,4] }
```

Motivation for Lua

- DEL users began to ask for more power, e.g. control flow (with conditionals and loops)
- SOL implementation finished, but not delivered, as support for procedural programming was soon to be required
- **Conclusion**: replace both SOL and DEL by a single, more powerful language

- **Existing Alternatives**
  - Tcl: "unfamiliar syntax", bad data description support, Unix only
  - Lisps: "unfriendly syntax"
  - Python: still in its infancy

No match for the free, do-it-yourself atmosphere at Tecgraf

Birth of Lua

- "Lua"—"moon" in Portuguese
  - cf. "SOL"—"sun" in Portuguese
- SOL's syntax for record and list construction

```
T = @track{ y=9, x=10, id="1992-34" }
```

valid in both SOL and Lua.

- Semantics differ:
  - tables represent both records and lists;
  - `track` (here) does not name a record type, it names a function to be applied.

Lua Feature Evolution

- Lua designers have shown good judgement.
- Learn **PL design** by asking:
  - What features were added to Lua and why?
  - What features were turned down and why?
- Learn **PL implementation** by asking:
  - How were the features implemented?
  - What kind of implementations were not possible due to
  - other implementation choices?

Lua Types

- Lua's type selection has remained fairly stable.
  - initially: numbers, strings, tables, nil, userdata (pointers to C objects), Lua functions, C functions
  - unified functions in v3.0; booleans and threads in v5.0
- **Tables:** any value as index
  - early syntax: @(), @[1,2], @{x=1,y=2}
  - later syntax: {}, {1,2}, {x=1,y=2}, {1,2,x=1,y=2}
    - sparse arrays OK: {[1000000000]=1}
  - element referencing sugar: a.x for a["x"]
  - tables with named functions for OO
    - for inheritance, define a table indexing operation

Tables

- The syntax of tables has evolved, the semantics of tables in Lua has not changed at all:
  - tables are still associative arrays and can store arbitrary pairs of values
- Effort in implementing tables efficiently
  - Lua 4.0, tables were implemented as pure hash tables, with all pairs stored explicitly
  - Lua 5.0, a hybrid representation for tables: every table contains a hash part and an array part, and both parts can be empty. Tables automatically adapt their two parts according to their contents.

Extensible Semantics

- Goals
  - allow tables to be used as a basis for objects and classes
- *fallbacks* in Lua 2.1(备选)
  - One function per operation (table indexing, arithmetic operations, string concatenation, order comparisons, and function calls) 当操作被应用到错误的值时，调用备选函数
- *tag methods* in Lua 3.0
  - tag-specific fallbacks, any value taggable
- *metatables* and *metamethods* in Lua 5.0

```
x = {}
function f () return -5 end
setmetatable(x, { __unm = f })
return -x --> -5
```

Expressing OOP Concepts

```
1  A = {}
2  A["b"] = 0
3  A["w"] = function(v)
4      A["b"] = A["b"]-v
5  end
6
7  A["w"](100.0)
```

(a) class A with two members

```
1  A = {b = 0}
2  function A.w(v)
3      A.b = A.b - v
4  end
5
6  a = A
7  a.w(100.0)   ✔
8  A = nil
9  a.w(100.0)   ✘
```

(b) Syntactic sugar of (a)

```
1  A = {b = 0}
2  function A.w(self,v)
3      self.b = self.b - v
4  end
5
6  a = A
7  a.w(100.0)   ✔
8  A = nil;
9  a.w(100.0)   ✔
```

(c) class A is singleton

```
1  function A.new(b)
2    return {b = b}
3  end
4
5  a = A.new()
6  A.w(a, 5)
7  --replaced with a.w(a,5) or a:w(5)
```

(d) Add new to make instances

Expressing OOP Concepts

```
1  LA = {}
2
3  for k,v in pairs(A) do
4    LA[k] = v
5  end
6
7  function LA.new()
8    local a = A.new()
9    a.l = 100
10   return a
11 end
12
13 function LA:w(v)
14   if v-self.b >= self.l then
15     error "Insufficient"
16   end
17   self.b = self.b - v
18 end
19
20 local a = LA.new()
21 LA.w(a, 5)
```

(e) Inheritance through class tables

```
1  function inherit(t)
2    local new_t = {}
3    for k, v in pairs(t) do
4      new_t[k] = v
5    end
6  end
```

(h) Generic inheritance via a table copy


```
1  A = {}
2  function A.new(b)
3    local a = {b = b}
4    for k,v in pairs(A) do
5      a[k] = v
6    end
7    return a
8  end
9  function A:w(n)
10   self.b = self.b - n
11 end
12 local a = A.new(500)
13 a:w(5)
14
15 LA = {}
16 function LA.new(b,l)
17   local a = {b = b}
18   a.l = l
19   for k,v in pairs(LA) do
20     a[k] = v
21   end
22   return a
23 end
24 function LA:w(n)
25   if n-self.b>=self.l then
26     error "Insufficient"
27   end
28   self.b = self.b - n
29 end
30 local a = LA.new(50,10)
31 a:w(5)
```

(f) "Normal" OOP

Overhead issue!

According to the definition, each instance of an account contains an entry for every method member, which leads to a lot of pointers, and a lot of overhead.

Assume you have a class with 30 methods, then every time you make an instance of the class, you have to allocate 30 strings and store them all in a table.

Expressing OOP Concepts

```
1  A = {b = 0}
2  function A:new(t)
3    t = t or {}
4    setmetatable(t,{__index = self})
5    return t
6  end
7
8  function A:w(n)
9    self.b = self.b - n
10 end
11
12 LA = A:new({l = 10})
13 function LA:w(v)
14   if v-self.b>=self.l then
15     error "Insufficient"
16   end
17   A.w(self, v)
18 end
19
20 a = LA:new({b = 50, limit = 10})
21 a:w(30)
22 a:w(30)   -- Insufficient
```

(g) Prototype-based objects

Use metatables to add a layer of indirection and to provide dynamic lookup on the metatable.

A group of related tables may share a common metatable (which describes their common behavior).
Line 3 creates object if user does not provide one; line 4 calls setmetatable to set or change the metatable of any new object t, and make t inherit its operations from the A table itself using the index metamethod, accordingly reducing the overhead mentioned before.

Expressing OOP Concepts

```
1  A = {b = 0}
2  function A:new(t)
3    t = t or {}
4    setmetatable(t,{__index = self})
5    return t
6  end
7
8  function A:w(n)
9    self.b = self.b - n
10 end
11
12 LA = A:new({l = 10})
13 function LA:w(v)
14   if v-self.b>=self.l then
15     error "Insufficient"
16   end
17   A.w(self, v)
18 end
19
20 a = LA:new({b = 50, limit = 10})
21 a:w(30)
22 a:w(30)   -- Insufficient
```

(g) Prototype-based objects

The derived class LA is just an instance of A but extended with member l.
LA inherits new from A. When new at line 20 executes, the self parameter will refer to LA. Therefore, value at index index in the metatable of a will be LA. Thus a inherits from LA, which inherits from A. When calling a:w at line 21, Lua cannot find a w field in a, so it looks into LA and there it finds the implementation for LA:w.

Scope, Function Calls and Storage Management

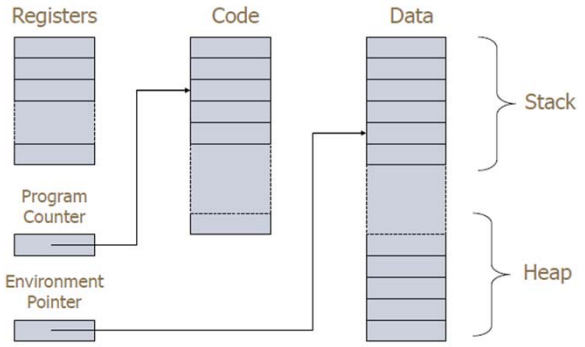
"Concepts in Programming Languages"
- Chapter 7: Scope, Functions, and Storage Management

Scope

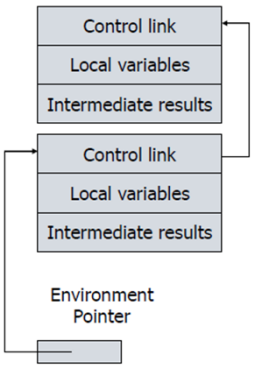
- Nested blocks, local variables
- Storage management
  - Enter block: allocate space for variables
  - Exits block: some or all space may be deallocated
- Static (lexical) scoping (Lua, etc.)
  - Global refers to declaration in closest enclosing block
- Dynamic scoping
  - Global refers to most recent activation record

```
local z = 0
if true then
   local z = 1
   print(z)
   z = 2
   print(z)
end
print(z)
```


Simplified Machine Model

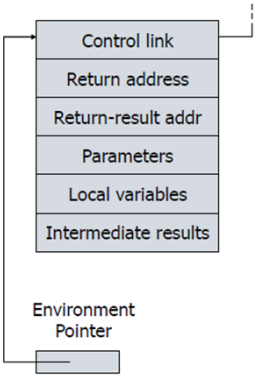


Activation record for in-link block



- Control link
  - Pointer to previous record on stack
- Push record on stack
  - Set new control link to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer

Activation record for function

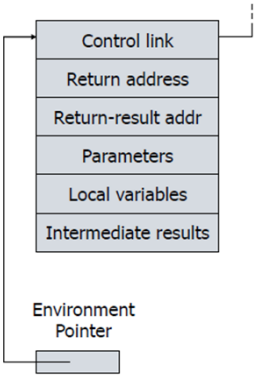


- Return address
  - Location of code to execute on function return
- Return-result address
  - Address in activation record of calling block to store function return val
- Parameters
  - Locations to contain data from calling block

First-order functions

- Parameter passing
  - pass-by-value: copy value to new activation record
  - pass-by-reference: copy ptr to new activation record
- Access to global variables
  - global variables are contained in an activation record higher “up” the stack
- Tail recursion
  - an optimization for certain recursive functions

Activation record for static scope



- Control link
  - Link to activation record of previous (calling) block
- Access link
  - Link to activation record of closest enclosing block in program text
- Difference
  - Control link depends on dynamic behavior of prog
  - Access link depends on static form of program text

Higher-order functions

- Language features
  - Functions passed as arguments
  - Functions that return functions from nested blocks
  - Need to maintain environment of function

  Functions as first class values
- Simpler case
  - Function passed as argument
  - Need pointer to activation record “higher up” in stack
- More complicated second case
  - Function returned as result of function call
  - Need to keep activation record of returning function

Closures

- Function value is pair
  - closure = < env, code >
- When a function represented by a closure is called,
  - Allocate activation record for call (as always)
  - Set the access link in the activation record using the environment pointer from the closure

Closures via "Upvalues"

- Lua authors wanted lexical scoping (词法作用域/静态作用域) early on
  - difficult due to technical restrictions
    - wanted to keep a simple array stack for activation records
    - one-pass compiler
- Lua 3.1 with a compromise called *upvalues*
  - In creating a function, make (frozen) copies of the values of any external variables used by a function.

```
function f ()     高阶函数：void →(void→int)
  local b = 1
  return (function () return %b + 1 end)  // b是外部的局部变量, upvalue
end
return f()() --> 2        upvalue 有些像C的static局部变量
```


Full Lexical Scoping

- Lua 5.0 got the real thing
- Solution: “Keep local variables in the (array-based) stack and only move them to the heap if they go out of scope while being referred by nested functions." (JUCS 11 #7)

```
function f ()
  local b = 1
  local inc_b = (function () b = b + 1 end)
  inc_b()
  return (function () return b end)
end
return f()() --> 2       closure: 一个匿名函数加上其可访问的upvalue
```


Tail Calls

- tail calls supported since 5.0
  - called function reuses the stack entry of the calling function
    - erases information from stack traces
- only for statements of the form `return f(...)`
  - `return n * fact(n-1)` does not result in a tail call

Coroutines

- *coroutines*—a general control abstraction
  - term introduced by Melvin Conway in 1963
  - has lacked a precise definition, but implies “the capability of keeping state between successive calls"
- have not been popular in mainstream languages
- classification:
  - *full coroutines* are stackful, and first-class objects
    - stackful coroutines can suspend their execution from within nested functions
  - an *asymmetric coroutine* is “subordinate” to its caller—can yield, caller can resume

Coroutines in Lua

- constraints: portability and C integration
  - cannot manipulate a C call stack in ANSI C
  - impossible: first-class continuations (as in Scheme), symmetric coroutines (e.g., in Modula-2)
- Lua 5.0 got *full asymmetric coroutines*, with `create`, `resume` and `yield` operations
  - ...and PUC-Rio guys gave proof of ample expressive power
  - capture only a partial continuation, from `yield` to `resume` — cannot have C parts there

  协程有现场保护

Coroutine Example

```
> return (string.gsub("abbc", "b",
        function (x) return "B" end))
aBBc
> return (string.gsub("abbc", "b",
        coroutine.wrap(function (x)
                coroutine.yield("B")
                coroutine.yield("C")
                end)))
aBCc
```


Embedding Lua

- Programming in Lua **Part IV • The C API**
  - ★ 24 – An Overview of the C API
  - 25 – Extending your Application
  - 26 – Calling C from Lua
- http://lua-users.org/lists/lua-l/2007-11/msg00248.html

Lua is designed for embedding

- Interpreter is small
  - Lua: 35 files, 14K LOC, 700 KB, ~1s compile time on Will's laptop
  - Python: 4000 files, 900K LOC, 300 MB, ~1.5m compile time
- Language is small
  - Not much syntax
  - Few core language concepts (e.g. no classes)
  - Small standard library
- Relatively simple C API
  - Small language = small API surface
  - Easily sandboxed

C embedded in Lua?

- Easy access to many libraries
  - C ABI is the lowest common denominator for many PLs
- Improve performance of bottlenecks
  - In Python, numpy and stdlib
- Extend language semantics
  - Async I/O, threading, …

Issues of Interop between PLs

- Lua and C
  - Type system: dynamic vs. static
  - Memory management: automatic vs. manual
- C and Fortran
  - Fortran: HPC (complex number) , column-major order, subroutine and function, call-by-reference
  - C: argc and argv , dynamic memory management, row-major order, call-by-value
  - E.g. generate mpif.h by wrapping mpi library written in C

Issues of Interop between PLs

- Spark
  - Write applications in Scala, Java, Python, R
  - Combine SQL, streaming and complex analytics (GraphX, MLlib, …)
  - Run on Hadoop, Mesos, Kubernetes, standalone, or in the cloud
  - Access diverse data sources, e.g. HDFS, Cassandra, HBase, Hive, Tachyon, Amazon S3
- Javascript and WebGL …
  - Zhen Zhang. xWIDL: Modular and Deep JavaScript API Misuses Checking Based on eXtended WebIDL, ACM SIGPLAN 2016 Student Research Contest, (Poster)

C API

- Issues
  - Static typing in C vs. Dynamic typing in Lua
  - Manual memory management in C vs. GC in Lua

  =>Use an abstract stack to exchange data between C & Lua

```
function foo(t) return t.x end

void foo_l(void) { /* Lua 1.0 */
  lua_Object t = lua_getparam(1);
  lua_Object r = lua_getfield(t, "x");
  lua_pushobject(r);
}

int foo_l(lua_State* L) { /* Lua 4.0 */
  lua_pushstring(L, "x");
  lua_gettable(L, 1);
  return 1;
}
```


Interop between Lua and C

- Two modes of use: Lua in C vs. C in Lua
- Lua uses a stack as an API
  - Prevent user from having to manage memory
  - Provide easy way to manage variadic inputs/outputs
  - Contrasts with Python manual refcounting
- Userdata provides a means of giving C values to Lua
- Lua as a configuration language: extending your app.