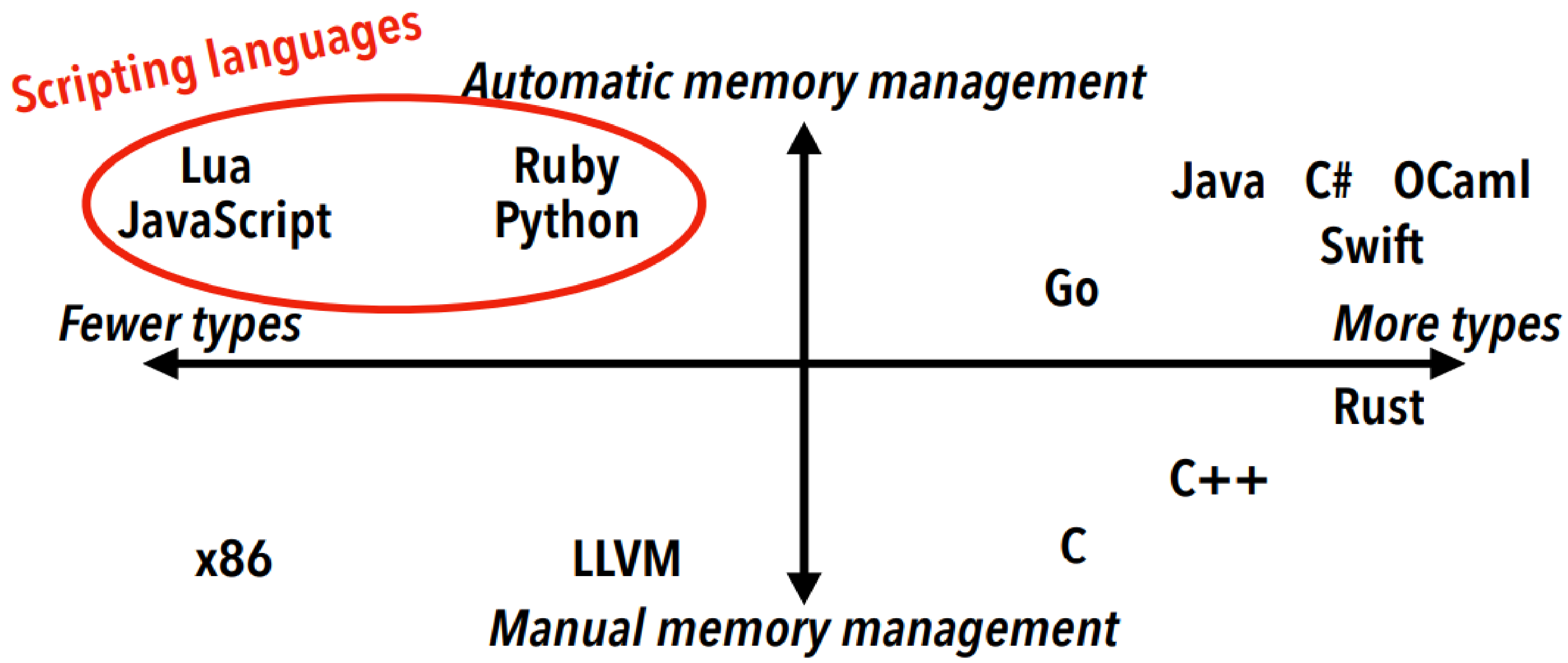
# Some PLs: Lua

Yu Zhang

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

# Scripting Languages

- 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! 无需承诺：先声明再使用

- 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 5.4: Jun 29, 2020, Lua 5.4.4: Jan 26, 2022
  - Smallish, e.g. Lua 5.1(Feb 2006) 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                          -- Modula-2
function fact(n)                PROCEDURE Fact(n: CARDINAL):
                                     CARDINAL;
  local a = 1                     VAR a: CARDINAL;
                                BEGIN
                                  a := 1
  for i = 1, n do                 FOR i := 1 TO n DO
    a = a * i                       a := a * i;
  end                             END;
  return a                        RETURN a;
end                             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

```
A = {}
A["b"] = 0
A["w"] = function(v)
    A["b"] = A["b"]−v
end

A["w"](100.0)
```

(a) class A with two members

```
A = {b = 0}
function A.w(v)
    A.b = A.b − v
end

a = A
a.w(100.0)    ✔
A = nil
a.w(100.0)    ✘
```

(b) Syntactic sugar of (a)

```
A = {b = 0}
function A.w(self,v)
    self.b = self.b − v
end

a = A
a.w(100.0)    ✔
A = nil;
a.w(100.0)    ✔
```

(c) class A is singleton

```
function A.new(b)
  return {b = b}
end

a = A.new()
A.w(a, 5)
−−replaced with a.w(a,5) or a:w(5)
```

(d) Add new to make instances

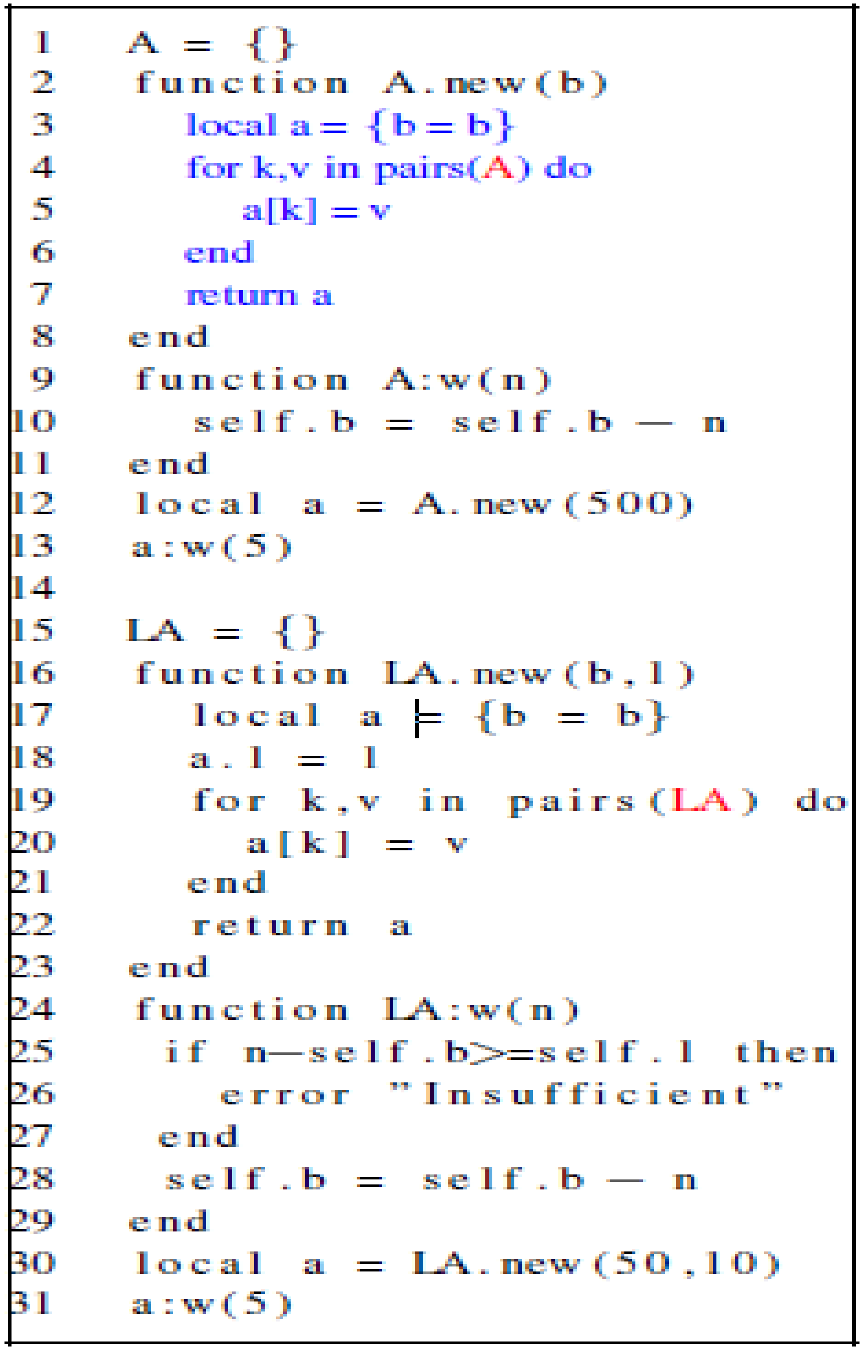
# Expressing OOP Concepts

```
LA = {}

for k,v in pairs(A) do
   LA[k] = v
end

function LA.new()
   local a = A.new()
   a.l = 100
   return a
end

function LA:w(v)
   if v-self.b >= self.l then
      error "Insufficient"
   end
   self.b = self.b - v
end

local a = LA.new()
LA.w(a, 5)
```

(e) Inheritance through class tables

```
function inherit(t)
   local new_t = {}
   for k, v in pairs(t) do
      new_t[k] = v
   end
end
```

(h) Generic inheritance via a table copy



```
A = {}
function A.new(b)
   local a = {b = b}
   for k,v in pairs(A) do
      a[k] = v
   end
   return a
end
function A:w(n)
   self.b = self.b − n
end
local a = A.new(500)
a:w(5)

LA = {}
function LA.new(b,l)
   local a = {b = b}
   a.l = l
   for k,v in pairs(LA) do
      a[k] = v
   end
   return a
end
function LA:w(n)
  if n−self.b>=self.l then
    error "Insufficient"
  end
  self.b = self.b − n
end
local a = LA.new(50,10)
a:w(5)
```

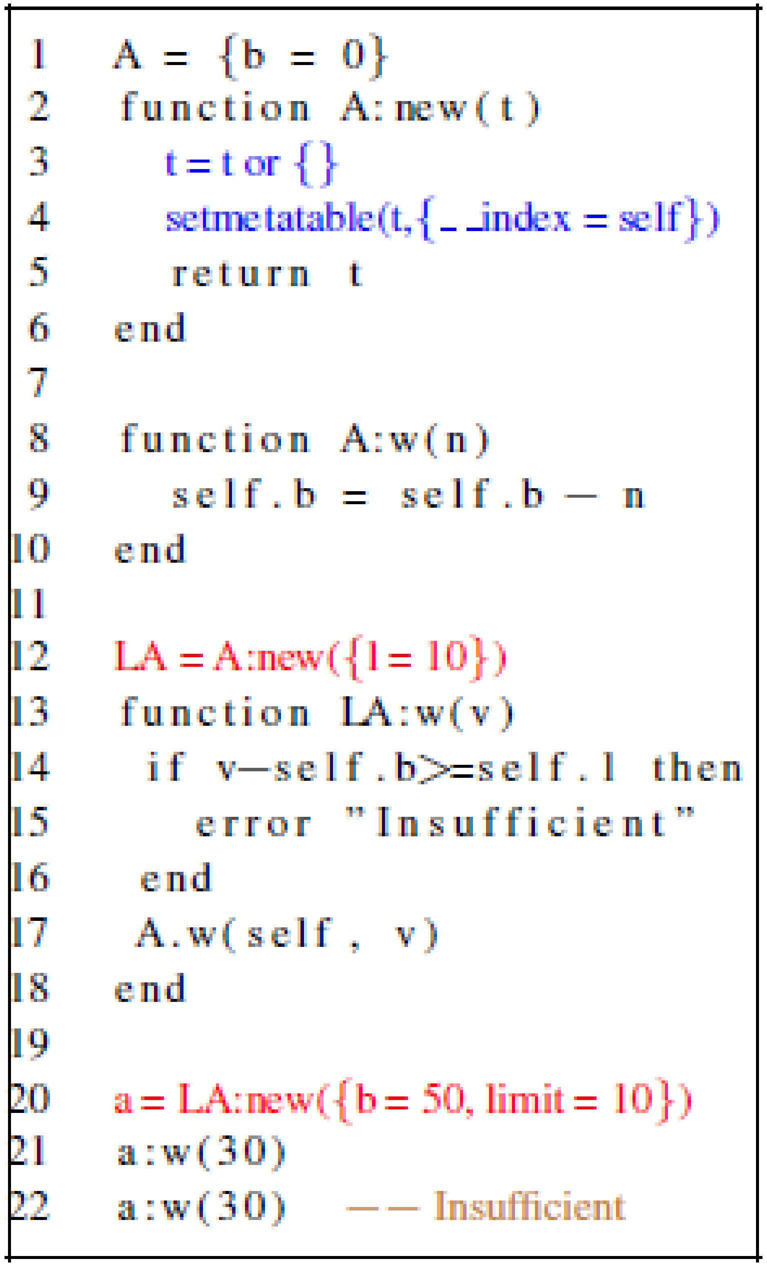
(f) "Normal" OOP

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



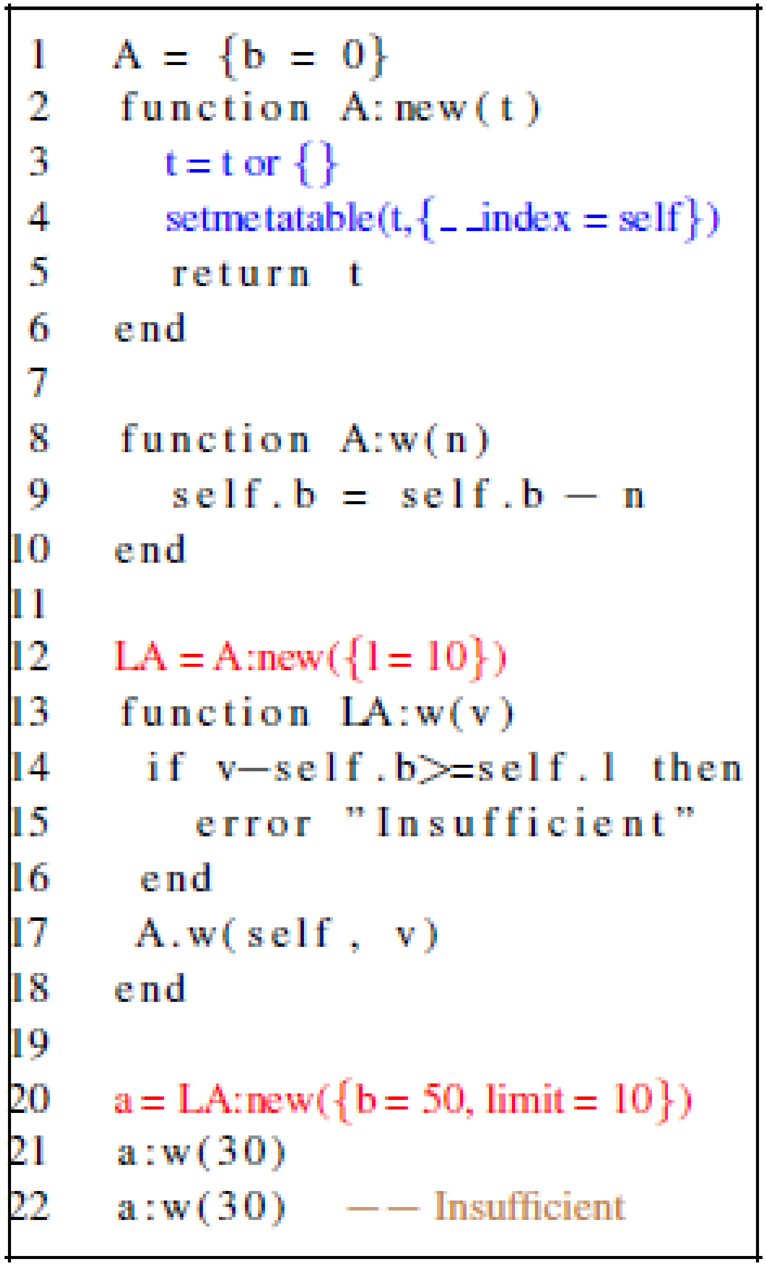
```
1   A = {b = 0}
2   function A:new(t)
3      t = t or {}
4      setmetatable(t,{__index = self})
5      return t
6   end
7
8   function A:w(n)
9      self.b = self.b − n
10  end
11
12  LA = A:new({l = 10})
13  function LA:w(v)
14    if v−self.b>=self.l then
15       error "Insufficient"
16    end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

## THANKS