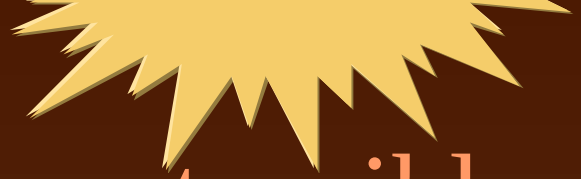


Lua - an extensible extension language

and

LuaOrb - and ORB for Lua

Pontifical Catholic University
Rio de Janeiro

http://www.tecgraf.puc-rio.br/lua/

Outline of the presentation

- Introduction (why Lua)
- The Lua language
  - Lua basics
  - Using Lua
  - *“Advanced tricks”* with Lua
  - Integrating Lua and C/C++
- Lua in serious applications
- LuaOrb
- Conclusion

Why Lua

- Scripting languages are necessary
- Can be used for configuration
- Configuration
  - can be expressed as parameters
  - users demand more
- A language with flow-control constructs is needed

Embedded languages

- Users want to customize their applications more and more
- Users are often partially-skilled programmars
- A new architecture arises : an application consisting of a *kernel* and a *configuration*.

Lua is such a configuration language

Lua basics

- Lua is an ***extensible extension*** language
- Lua is a “Pascal-like” interpreted compiled language
- Lua has dynamic typing
- Only a few basic types: number, string, nil, table and function.
- A function is a *first-class* value.

Simple Lua examples

Lua as a configuration language

```
width = 420
height = width*3/2
color = “blue”
```

Lua as a configuration language

```
function Bound(w,h)
  if w<20 then w=20
  elseif w>500 then w=500
  end
  local minH = w*3/2
  if h < minH then h = minH end
  return w, h
end
width, height = Bound(420, 500)
if monochrome then color = “black”
else color = “blue” end
```

Tables (associative arrays)

- can be indexed with values of any type
- a simple mechanism
- but implements the concept of
  - records
  - arrays
  - recursive data types (pointers)

Example - tables as a *linked list*

```
list={}
current = list
i = 0
while i<10 do
  current.value = i
  current.next = {}
  current = current.next{}
  i=i+1
end
current.value=i
current.next=list
```

Note:

`a.name`

is syntactical sugar for

`a[“name”]`

Tables: construction

- `list = {}`
- `window = { x = 200, y = 300, foreground = "blue" }`
- `window1 = Window{ x = 200, y = 300, foreground = "blue" }`
  ... equal to: `Window({...})`
- `colors = {"blue", "red"}`
  equal to:
  `colors = {}; colors[1]= "blue" colors[2]= "red"`

Another example: clone

```
function clone(o)
  local newO = {}
  local i, v = next(o,nil)
  while i do
    newO[i] = v
    i, v = next(o,i)
  end
  return newO
end
```

Another example: clone as ***deep copy***

```
function clone(o)
  local newO = {}
  local i, v = next(o,nil)
  while i do
    if type(v)="table” then v=clone(v) end
    newO[i] = v
    i, v = next(o,i)
  end
  return newO
end
```

Iterating global varialbles: `save` and `restore`

```
function save()
  local env = {}
  local n, v = nextvar(nil)
  while n do
    env[n] = v
    n, v = nextvar(i)
  end
  return env
end
```

```
function restore(env)
  -- save builtin functions
  local nextvar, next, setglobal = \
      nextvar, next, setglobal
  -- erase all global variables
  local n, v = nextvar(nil)
  while n do
    setglobal(n,nil)
    n, v = nextvar(i)
  end
  -- restore old values
  n, v = next(env,nil)
  while n do
    setglobal(n, v)
    n, v = next(env, n)
  end
end
```

Support for object-oriented programming

- Support at syntactic level:

```
function object:method(params)
```

is equal to:

```
function dummy_name(self,params)
      ...
end

object.method = dummny_name
```

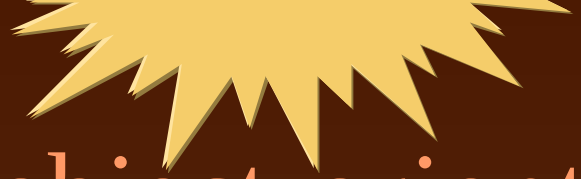
Support for object-oriented programming (cont.)

- Support at syntatic level:

  ```
  receive:method(params)
  ```

  is translated to:

  ```
  receiver.method(receiver,params)
  ```



Support for object-oriented programming (cont.)

But where are the basic concepts of OOP?

inheritance???

Solution: Fallbacks

- Lua provides *fallbacks*: general purpose exception handlers
- Support the following exceptions:
  - "arith", "order", "concat" - for overloading
  - "index" - for inheritance, but also many others
  - "gettable", "settable”
  - "function”
  - "gc" - when the garbage collector is to be run

Inheritance

```
function Inherit (object, field)
  if (field == "parent" then  -- avoid loops
    return nil
  end
  local p = object.parent
  if type(p) == "table” then
    return p[field]    -- this may recurse
  else
    return nil
  end
end
```

Inheritance

- with fallbacks, we can get exactly the kind of inheritance we want
- multiple inheritance
- or *double inheritance* - an inheritance with a limited number of parents:

  a *parent* and a *godparent*:

```
a = {parent = a1, godparent = {
  parent = a2, godparent = a3}}
```

Integrating Lua and C

```
char* getenv(char*);
```

So, the appropriate functions to call are lua_isstring, lua_getstring and lua_pushstring:

```
void wrap_getenv(void)
{
 lua_Object o=lua_getparam(1);
 if (lua_isstring(o))
  lua_pushstring(getenv(lua_getstring(o)));
 else
  lua_error("string expected in argument #1
to getenv");
}
```

Integrating Lua and C

Since version 3.0, Lua provides better mechanisms.

```
void wrap_getenv(void)
{
  lua_pushstring(getenv(
      luaL_check_string(1)));
}
```

Function is registered by calling

```
lua_register("getenv",wrap_getenv)
```

In fact, this is almost exactly what the standard library does.

Integrating Lua and C++

Lua file

```
name = "lua”
function display (name) print("C API:"..name)
end
```

C++ file

```
#include "lua++.h"
LuaReference ref;
LuaValue(change).store("change");  //
Register function
Lua::record("change",change); // or this way
Lua::dofile("test.lua"); // Execute file
LuaObject name = Lua::getglobal("name");
LuaObject display=Lua::getglobal("display");
display(name);
```

Lua in serious applications

- TeCGraf *(Tecnologia em Computaci Grafica)* is using Lua in several applications:
  - Configurable report generator for lithology profiles
  - Storing structured graphical metafiles
  - High level, generic graphical data entry
  - Generic attribute configuration for finite element meshes

LuaOrb

an Orb for Lua

An introduction of CORBA

Sorry, just joking

LuaOrb

- LuaOrb is a library providing Lua access to CORBA objects
- LuaOrb allows Lua to call methods on CORBA objects:

IDL:

```
struct book { string author; string title;};
interface foo { void add_book(book abook);};
```

Lua client:

```
a_foo = createproxy("foo")
a_book = {author = "Mr. X", title="NewBook"}
a_foo:add_book(a_book)
```

IDL to Lua mapping

- Simple and invisible
- When an Lua object (table) is passed as an parameter, it is checked, whether it complies to the IDL specification
- Minor changes to an interface do not affect the Lua code:
  - number type changes
  - changing between a sequence and an array
  - attribute deletion

IDL to Lua mapping: Server objects

- As easy and simple as parameter passing:

```
interface listener {
  oneway void put(long i);
}
interface generator {
  void set_listener(listener r);
}

l = { put = function (self,i) print(i) end}
gen = createproxy("generator")
gen:set_listener(l)
```

DSI in LuaOrb

- Lua is allowed to access the IR
- DSI is in Lua still *very simple*
- Allows remote installation of new methods

DSI in Lua (cont. -- IDL code)

```
interface LuaDSIObject {
  readonly attribute Object obj;
  void InstallImplementation(string opname,
      string luaCode);
}

interface ServerLua {
  LuaDSIObject Instance(
      CORBA::InterfaceDef intf);
}
```

DSI in Lua (cont. -- Lua Code)

```
-- creates a new object with type "listener"
l = Server:Instance(IR:lookup("listener"))
-- creates a new method called 'put'
l:InstallImplementation("put","function
    (self, i) print(i) end")
gen = createproxy("generator")
gen:set_listener(l.obj)
```

Lua and Interface repository

- Lua is able to access the IR
- Lua is also able to dynamically update the IR -- adding record for new services, or their new versions
- The possible features are unforeseen

Conclusion

- Lua is a simply, yet powerful language
- Lua is suitable to be used as an embedded language (it’s interpreter is small, implemented as a portable C library)
- Lua *can* be used as a standalone language
- Lua has been successfully comercially used
- There’s a number of extensions for Lua: CGILua, LuaOrb, LuaJava, Lua-PSQL, TkLua,...
- Lua really is an *extensible extension* languae

Add on 1: Typechecking in Lua

```
TNumber="number"
TPoint={x=TNumber, y=TNumber}
TColor={red=TNumber, blue=TNumber,
green=TNumber}
TRectangle={topleft=TPoint, botright=TPoint}
TWindow={title="string", bounds=TRectangle,
color=TColor}
```

Given such descriptions, the following function checks whether a value has a given type:

```
function checkType(d, t)
  if type(t) == "string" then
-- t is the name of a type
    return (type(d) == t)
  else
-- t is a table, so d must also be a table
    if type(d) ~= "table" then
      return nil
    else
-- d is also a table; check its fields
      local i,v = next(t,nil)
      while i do
        if not checkType(d[i],v) then
          return nil
        end
        i,v = next(t,i)
      end
    end
  end ; return 1
```

Add on 2: A self-reproducing program in Lua

```
y = [[ print("y = [[" .. y                    \
    .. "]]\ndostring(y)") ]]
dostring(y)
```

The End

Please wake up now...