The benefits of small programming language

• Ease of use
• Methodology
• Elegance
• Simplified implementation
• Do not suffer from things you do not use
Lua

• A scripting language
• Simplicity as one of its main goals
  – small size too
• Tricky balance between “as simple as possible” x “but not simpler”
• Many users and uses
Lua Introduction

- Fast interpreted scripting language
- Offers concurrency via coroutines
- Offers support for object-oriented programming
- Small, and therefore easily embeddable
- Powerful but simple
- Enjoys great popularity in the video game industry, in particular for the popular role playing game World of Warcraft
- Other adapters: Adobe, Wikipedia
- Lua translates to “Moon” in Portuguese
The Engine Survey (03/02/09, Gamasutra):
What script languages are most people using?

- Lua: 50.0%
- Custom Language: 40.0%
- C/C++ Variant: 30.0%
- Unreal Script: 20.0%
- Python: 10.0%
Adobe Lightroom

One million lines of Lua code
Troy Gaul. Lightroom Exposed.
http://www.troygaul.com

Lua 63%
C++ 16%
C 9%
Obj 12%
History of Lua (1)

- Designed by Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo at the Pontifical Catholic University of Rio de Janeiro, Brazil
- First appeared in 1993
- Influenced by Scheme, SNOBOL, Modula, CLU and C++
- Lua was initially developed as a result of strict trade barriers instituted by the Brazilian government
The intended audience was not professional programmers, so the designers wanted to avoid cryptic syntax and semantics.

Lua was born from SOL (Simple Object Language) and DEL (Data-Entry Language).

- SOL and DEL had a lack of flow control
Lua is an Embedded Language

- Embedded in C, C++, C#, Java, Ruby, Python, …
- But no need for ‘eval’
  - Tedious
  - Requires serialization
- Interface to communicate with the host language
- Efficient transfer of data and control
void copy (int ar[], int n) {
    int i;
    eval("ar = {}”); /* create an empty array */
    for (i =0; i <n; i++){
        char buff[100];
        sprintf(buff, “ar[%d] = %d”, i + 1, ar[i]);
        eval(buff); /* assign i-th element */
    }
}
Lua Overview

• Semantically somewhat similar to Scheme
• Similar to JavaScript, too
• Dynamically typed
• All objects have unlimited extent
  – incremental garbage collector
• Functions are first-class values with static scoping
• Proper tail recursive
Lua main goals

- Simplicity
- Small size
- Portability
- Embedability
  - Scripting language
function fact (n)
    local f = 1
    for i=2,n do f = f * i; end
    return f
end

↓

fact = function (n)
    local f = 1
    for i=2,n do f = f * i; end
    return f
end
• Variable Example

\[
x = 10 \quad \text{(Global)}
\]

local \( i = 1 \)

while \( i \leq x \) do
    local \( x = i \times 2 \)
    print(\( x \))
    \( i = i + 1 \)
end

Output: 2, 4, 6, 8 ...
Like most languages, Lua offers small and conventional set of control structures

- Conditionals
  - if statements
- Iteration
  - while
  - repeat
  - for
Control Structures (2)

- **if Statement**
  
  ```
  if a <= 0 then
    a = a + 1
  end

  if a < 0 then
    a = a + 1
  elseif a > 0 then
    a = a - 1
  else
    a = 0
  end
  ```

  Simple if statement

  Nested if-else statement
Conditional Structures (3)

• While Loop

Lua first tests the while condition, if it is false then the loop ends, otherwise it executes the body

```lua
a = {1,2,3,4,5}
i = 1
while a[i] do
    print(a[i])
    i = i + 1
end
```

Output: 1,2,3,4,5
• Repeat

Repeat-until statement repeats its body until the condition is true. This ensures that the body of the loop is executed at least once.

```plaintext
a = 0
repeat
  a = a + 1
  print(a)
until a > 10
```

Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
Control Structures (5)

• Numeric For Loop

```
for i=1,10,1 do
    print(i)
end
```

Output: 1,2,3,4,5,6,7,8,9,10
Control Structures (6)

• Generic For Loop
  – Allows you to traverse all values returned by an iterator function

```lua
colors = {“red”, “green”, “blue”}

for i,v in ipairs(colors) do
  print(i,v)
end
```

**Output:** 1 red, 2 green, 3 blue
• Tables
• Coroutines
• The Lua API
Tables

• Associative arrays
  – Any value is a key
• Only data-structure mechanism in Lua
• Efficiently represent other data structures
• Tables implement most data structures in a simple and efficient way
• records: syntactical sugar t.x for t["x"]

t = {}
t.x = 10
t.y = 20
print(t.x, t.y)
print(t["x"], t["y"])
• **Arrays**: integers as indexes

```plaintext
a = {}
for i=1,n do a[i] = 0 end
```

• **Sets**: elements as indices

```plaintext
t = {}
t[x] = true  ---  t = t ∪ {x}
if t[x] then  ---  x ∈ t ?
...
```
Modules

• Tables populated with functions
  
  ```
  local math = require "math"
  print(math.sqrt(10))
  ```

• Several facilities come for free
  – Submodules
  – Local names
  
  ```
  local m = require "math"
  print(m.sqrt(20))
  local f = m.sqrt
  print(f(10))
  ```
Objects

• first-class functions + tables ≈ objects
• syntactical sugar for methods
  – handles self
• Tables in Lua are objects
  – They have states like objects
  – They have an identity, a selfness, like objects

Account = {balance = 0}
function Account.withdraw(amt)
    Account.balance = Account.balance - amt
end
Account.withdraw(19.99)
print(Account.balance) --> -19.99
A more flexible and familiar approach is to operate on the receiver itself

```plaintext
function Account.withdraw(self, amt)
    self.balance = self.balance - amt
end

a1 = Account
a1:withdraw(19.99) -- a1.withdraw(a1, 19.99)
print(a1.balance) --> -19.99
```
Object-Oriented Programming (3)

• Classes
  – Lua does not have the concept of class, however Lua can emulate classes via prototyping

```lua
function Account.new(self, object)
    object = object or {}
    setmetatable(object, self)
    self.__index = self
    return object
end
```

Makes the new object a prototype of Account
So now that we created the concept of an Account class we can create multiple instances of Account.

```ruby
a1 = Account.new(a1, {balance=0})
a2 = Account.new(a2, {balance=100.25})
Account.withdraw(a1, 100.00)
Account.withdraw(a2, 100.25)
print(a1.balance)  -->  -100.00
print(a2.balance)  -->  0.00
```
Functions

- Functions in Lua are anonymous
  ```lua
  function square(x)
    local sqr = 0
    sqr = x * x
    return sqr
  end
  ```

- The `square` function is called as we would expect
  ```lua
  print(square(2)) --> 4
  ```

- If a function only has one argument then it can be called without parenthesis
  ```lua
  result = square "2"
  print(result) --> 4
  ```
Functions (2)

- Closures
  - A function which returns a function
  - Has full access to local variables
  - This allows Lua to implement functional programming

```lua
function newCounter()
    local i = 0
    return function()
        i = i + 1
        return i
    end
end

function newCounter()
    local i = 0
    return function()
        i = i + 1
        return i
    end
end
```

```plaintext
function newCounter()
    local i = 0
    return function()
        i = i + 1
        return i
    end
end
```

```plaintext
c1 = newCounter()
prompt(c1()) --> 1
prompt(c1()) --> 2

c2 = newCounter()
prompt(c2()) --> 1
prompt(c1()) --> 3
prompt(c2()) --> 2
```
Coroutines are similar to threads but not exactly the same

- A program with threads can run several threads concurrently
- Coroutines are collaborative
  - A program with coroutines is running only one of its coroutines and this running coroutine only suspends its execution when it explicitly requests to be suspended
Coroutines (2)

• Creating a coroutine is straightforward
  
  ```
  co = coroutine.create(function()
    print(“hi”)
  end)
  ```

• Creating a coroutine returns a value type, thread

  ```
  print(co) -- thread: 0x8071d98
  ```

• Coroutines have three states: suspended, running, and dead. Upon creation the coroutine is in suspended state
Coroutines (3)

- To resume a coroutine we use the function `coroutine.resume`
  
  ```javascript
  coroutine.resume(co) --> hi
  ```

- When a coroutine terminates it is now in the dead state, from which it cannot return

- Using the `yield` function we can suspend the coroutine execution so it can be run later
  
  ```javascript
  co = coroutine.create(function()
  for i=1,10 do
    print("co",i)
    coroutine.yield()
    end
  end)
  ```

Suspends the coroutine
• If we resume the coroutine and then check its status we will see the `yield` function at work

```python
coroutine.resume(co)  --> co 1
print(coroutine.status(co))  --> suspended
```

• To finish the execution we have to continuously `resume` the coroutine

```python
coroutine.resume(co)  --> co 2
coroutine.resume(co)  --> co 3
...
coroutine.resume(co)  --> co 10
```

Coroutine has run to completion. It is now in a dead state
Coroutines
Producer-Consumer Example

• Producer
  – Produce items to be consumed by the consumer

```lua
function producer ()
    return coroutine.create(function ()
        item = 1
        while item < 10 do
            print("Producing item ", item)
            coroutine.yield(item)
            item = item + 1
        end
    end)
end
```

Produce an item
Suspend the producer
Coroutines
Producer-Consumer Example

- Consumer
  - Gets items from the producer

```lua
function consumer(producer)
    while true do
        local status, value = coroutine.resume(producer)
        if status == false then
            return
        end
        print("Consumed item ", value)
    end
end
```

Consume the item from producer
Check if the producer is dead
Coroutines
Producer-Consumer Example

• Launch the producer-consumer
  
  \[
  p = \text{producer}() \\
  \text{consumer}(p)
  \]

  Output:
  Producing item 1
  Consumed item 1
  Producing item 2
  Consumed item 2
  Producing item 3
  Consumed item 3
  ...
  Producing item 9
  Consumed item 9
  Consumed item nil
Coroutines in Lua

- First class values
  - May be invoked anywhere
- Stackful
- A coroutine can transfer control from inside any number of function calls
- Asymmetric
  - different commands to resume and to yield
Coroutines in Lua

- Simple and efficient implementation
- The easy part of multithreading
- first class + stackful = complete coroutines
- equivalent to one-shot continuations
- coroutines present one-shot continuations in a format that is more familiar to most programmers
Coroutines vs. continuations

- Most uses of continuations can be coded with coroutines
- "Who has the main loop" problem
- Producer-consumer extending x embedding
- Iterators x generators
- The same-fringe problem
- Collaborative multithreading
Garbage Collection

- Lua performs automatic memory management
- Users do not have to worry about allocating and freeing memory
- This garbage collector will routinely look for dead objects
  - objects no longer accessible by Lua
The Lua API

- Lua is a library
- formally, an ADT (a quite complex one)
  - 79 functions
- the entire language actually describes the argument to one function of that library load
  - load gets a stream with source code and returns a function that is semantically equivalent to that code
Basic (Naive) Lua Interpreter

```c
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int main (int argc, char **argv) {
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);
    luaL_loadfile(L, argv[1]);
    lua_call(L, 0, 0);
    return 0;
}
```
Most APIs use some kind of “Value” type in C: PyObject (Python), jobject (JNI).
Problem: garbage collection.
Python: explicit manipulation of reference counts.
JNI: local and global references.
Easy to create dangling references and memory leaks.
The Lua-C API

- Lua API has no LuaObject type
- a Lua object lives only inside Lua
- two structures keep objects used by C:
  - the registry
  - the stack
The Registry

- Sometimes, a reference to a Lua object must outlast a C function
- NewGlobalRef in the JNI
- The registry is a regular Lua table always accessible by the API
  - no new concepts
  - to create a new “global reference”, store the Lua object at a unique key in the registry and keeps the key
The Stack

• keep all Lua objects in use by a C function
• *injection functions*
• convert a C value into a Lua value
• push the result into the stack
• *projection functions*
• convert a Lua value into a C value
• get the Lua value from anywhere in the stack
The Stack

- example: calling a Lua function from C
  - push function, push arguments, do the call, get result from the stack

/* calling f("hello", 4.5) */
lua_getglobal(L, "f");
lua_pushstring(L, "hello");
lua_pushnumber(L, 4.5);
lua_call(L, 2, 1);
if (lua_isnumber(L, 1))
    printf("%f\n", lua_getnumber(L, 1));
The Stack

• example: calling a C function from Lua
  – get arguments from the stack, do computation
  – push arguments into the stack

```c
static int l_sqrt (lua_State *L) {
    double n = luaL_checknumber(L, 1);
    lua_pushnumber(L, sqrt(n));
    return 1; /* number of results */
}
```
The Lua-C API: problems

- too low level
- some operations need too many calls
- stack-oriented programming sometimes is confusing
- what is where
- no direct mapping of complex types
- may be slow for large values
• any language design involves conflicting goals
• designers must solve conflicts consciously or not
• to get simplicity we must give something
  – performance, ease of use, particular features or libraries
Conclusion

• Lua is an elegant language, providing easy, readable code

• Lua can easily be extended using C or Lua itself

• Lua offers many of the features of much larger languages
  – Concurrency
  – Object-Oriented Programming

• Lua manual link
  – http://www.lua.org/pil/index.html#P1
Conclusions

• any language design involves conflicting goals
• designers must solve conflicts
• consciously or not
• to get simplicity we must give something
• performance, easy of use, particular features or libraries,
Conclusions

- simplicity is not an absolute goal
- it must be pursued incessantly as the language evolve
- it is much easier to add a feature than to remove one
- start simple, grow as needed
- it is very hard to anticipate all implications of a new feature
- clash with future features
Conclusions

• “Mechanisms instead of policies”
• e.g., delegation
• effective way to avoid tough decisions
• this itself is a decision...