A Short Lua Overview

Albrecht Zimmermann

20th April 2005

What is Lua

- Lightweight script language
- Extension to existing program in C to configure program, modify run-time behaviour
- *Embedded* in host program, not designed to be used stand-alone
- Dynamically typed

Syntactic Conventions

- Identifiers consist of digits, letters, underscore - cannot begin with digits
- 21 keywords, which are reserved, cannot be used as identifiers (the usual: *and, end, for* etc.)
- Case-sensitive: *and* is keyword, *And* identifier
- Identifiers starting with an underscore reserved for internal Lua variables

Types and Typing

- 8 basic types:
  - **nil** - type of the value *nil* which is different from any other value
  - **boolean** - has values *true, false*
  - **number** - double-precision floating point
  - **string** - array of characters
  - **function** - first-class value, means that functions can be stored in variables, passed as argument, returned
  - **userdata** - arbitrary C data, basically raw memory content, can be manipulated only through C API
  - **thread** - independent threads of execution
  - **table** - associative arrays, i.e. index does not have to be number, only structure, index can be used as `table.index`, can be nested

Types and Typing cont.

- Dynamic typing: type of variable depends on stored value
- Re-assignment of different value changes type
- Assignment with $=$
- Multiple assignments possible

Types and Typing - Examples

- a = 1 - type **number**
  a = "hello" - type **string**
  a = { item1="abc" } - type **table**, a[item1]=a.item1=abc
- x, y = 2, "there" - x=2, y="there"
  Expressions on the left and values on the right are evaluated before assignment
- String concatenation a = "hello".." world" - a = "hello world"
- x = function() print("hello") end - calling x() would now print `hello`
- Value *nil* means variable has no value and thus does not exist, checking whether a variable has the value *nil* is essentially existence check

Chunks and Blocks

- Chunk - list of statements, each *optionally* followed by a semicolon
- Sequentially executed, can be stored in files or as string in host program, have local scope, can define variables
- Block - Chunk, which may be delimited by **do** and **end**
- Helps to control variable scope

Control Structures

- **for** variable = start_val, stop_val [, step_size] **do** block **end**
- **for** variable [, variable] **in** expression-list **do** block **end**
  e.g. for table traversal:
  for key,value **in** table **do** print(key,value) **end**
- **while** expression **do** block **end**
- **repeat** block **until** expression
- **break** or **return** can be used to end loop
- Have to be last statement in block (followed by **end**), so if break somewhere in the middle wanted, explicit block
  do break **end**
- **if** expression **then** block {**elsif** expression **then** block}
  [**else** block] **end**

Functions

- Definition:
  - Name = **function** ([parameterlist]) block **end**
  - **function** Name ([parameterlist]) block **end**
  - **local function** Name ([parameterlist]) block **end** - this dies once the block is exited
- Call:
  - **function** foo(a,b,c) print(a,b,c) **end** - example
  - foo() - will print *nil nil nil*
  - foo(1,2,3,4) - will print 123
  - **function** foo(...) block **end** - variable list of parameters is put into table *arg*, table traversal methods used to access entries

Functions - cont.

- Return values:
  - foo() - no return value
  - x = foo() - adjusted to one return value
  - x, y = foo() - adjusted to two values
  - {foo()} - creates table of all returned values
  - Inside function: **return** [value] [, value]
  - For variable return list: **return** unpack(value-array)

Libraries

Lua includes libraries for

- Basic functions
- String manipulation
- Table manipulation
- Mathematical functions
- Input/Output methods
- Operating system facilities
- Debugging facilities

C API

- Declared in `lua.h`
- Whole state of a Lua interpreter (global variables, stack) stored in structure of type `lua_State`, has to be passed to each function in the library, except for lua_open, which returns such a pointer
- Create: `lua_State *lua_open(void);`, Delete: `void lua_close(lua_State *L);`
- Virtual stack used for communication with host program, each entry a Lua value (*nil, number,...*)
- Not real stack, reference to any stack entry possible
- Lua chunks loaded with `int lua_load`
- Calling functions: first push function on stack, then arguments in direct order, finally call `void lua_call`

C API cont.

Extending Lua with C functions:

- Must be of type `lua_CFunction`:
  ```
  typedef int (*lua_CFunction) (lua_State *L);
  ```
- Arguments are passed from Lua to the function in direct order on the stack
- Results are pushed by the function on the stack in direct order, function returns number of results
- For registering there is a macro

In-Depth Information

- Lua 5.0 reference:
  `http://www.lua.org/manual/5.0/manual.html`
- Lua Wiki including tutorials:
  `http://lua-users.org/wiki/`