1 Lua Basics

1.1 Introduction

Lua is a small language especially well suited for extending C/C++ applications to be scriptable. Its origins are in data definition language designed in 1993 for automating task of data input to simulations in Brazilian oil industry, but it has been used in numerous games in recent years.

1.2 Lexical conventions

Lexical conventions match closely to most other common languages. Identifiers and key words are case-sensitive. Identifiers need to begin with alphabetic character or underscore, and any alphanumeric character can follow. Only single-line comments are supported; comments start with $--$ (two dashes). String literals can be created with C-like " " pair and contain escape sequences like \n (newline). Lua supports also optionally multiline string literals in form of [[ ]] pair, for example:

```
a = [[ this is
multiline string
literal ]]
```

String literals created with [[ ]] pair can span multiple lines but escape sequences are not expanded inside the pair.

To conform Unix scripting conventions, if *chunk* starts with # character the first line is skipped. Chunk is the unit of execution in Lua – more about chunks in section 1.4.

1.3 Types and values

Lua is dynamically typed language, as most scripting languages are. Basic types in Lua are nil, boolean, number, string, table, function and userdata.

Boolean and number have value semantics. Strings have internally reference semantics, but strings are immutable, so they actually appear to user as having value semantics. Nil is special type which roughly matches C NULL value. Function instantiations, *closures*, are first-class variables in Lua and they have reference copy semantics, as well as tables and userdata.

Lua relies heavily on associative arrays, that is arrays which can be accessed by any type except nil. Associative arrays are implemented with *table* type in Lua. Tables can also be heterogeneous, they can contain values of all types except nil. Tables are the only data structuring mechanism in Lua. In addition to named record access using $array["name"]$, Lua provides syntactic sugar $array.name$

and iteration over table elements using $next(array, key)$ returns key and value of next array element.

Userdata is special data type which can be modified only from C code. This guarantees integrity when extending C application by scripting as the scripts cannot modify the value of userdata. Userdata has reference semantics, as do have tables and functions. *Metatables* can be used to define operations on userdata values, which allows Lua to be extended with custom types. Every table has metatable, which defines behaviour of the original table. For example in in addition Lua calls __add member of the metatable.

Lua is not object oriented language, but it does support object oriented programming by the usage of tables. Lua tables can contain associations to functions and the table itself can be passed as first parameter to the function by using : operator. For example $a : f()$ calls function f of table a and passes the table as an argument to the function.

Lua provides coercion between strings and numbers in run-time. This is applied so that any arithmetic operation tries to convert a string to number. Also when a string is expected a number is converted to string automatically.

1.4 Statements

As Lua was intended for extending applications, it has no concept of $main()$ function. Basic unit of execution is *chunk*. Chunk is sequence of Lua statements, which are optionally followed by semicolon. Chunks are interpreted as anonymouys function, so chunks can have local variables and chunks can return values.

In addition to traditional assignment, Lua supports assignment of multiple values. For example

```
a,b,c = 1,2,3
```

assigns a=1, b=2 and c=3. Parameters to assignment are evaluated applicatively so that

```
x,y = y,x
```

performs swap operation correctly between x and y.

Lua as controls structures similar to other common imperative languages. For example following code prints odd positive numbers below 100:

```
local n = 1
local odd = true
while n < 100 do
  if odd then
    print( n )
  end
  odd = not odd
  n = n + 1
end
```

In addition to $while$, Lua has $repeat$ and for loops. Repeat has the form

```
repeat block until exp
```

for loop has two forms, numerical and generic. Numeric form is

```
for i=first,increment,last do block end
```

and generic form is

```
for v1,...,vn in explist do block end
```

Generic for form is shorthand for the code below. Note that $v1, v2..., vn$ are all locals inside for loop.

```
do
  local _f, _s, v1 = explist
  local v2, ... , vn
  while true do
    v1, ..., vn = _f(_s, v1)
    if v1 == nil then break end
    block
  end
end
```

Lua has also $return$ and $break$ statements. $return$ is used to return value from block and $break$ is used to break iteration inside loop. Lua's $return$ statement supports multiple return values. The other difference to C language $return$ semantics is that $return$ and $break$ can only be executed as the last statement of their (inner) blocks.

1.5 Expressions

Lua expressions are similar to C expressions. In arithmetic operators the main difference is power operator $\wedge$, which is only present as pow standard library function in C. Relational operators are also similar to C, with the difference of inequality which is $\sim=$ in Lua. Logical operators are expressed in written form: and, not, or. Booleans are $true$ and $false$, but they are quite recent addition to Lua and so nil is still considered false as well as in versions preceding Lua 5.

String concatenation is done with .. operator. Due to coercion rules, numbers can be concatenated to strings with this as well.

Precedence order in Lua is similar to other common languages like C. Concatenation comes after arithmetic but before relational operators.

1.6 Functions

Lua functions are first-class variables with reference copy semantics. Function is defined by

```
function f( parameters )
   ...
end
```

This is actually syntactic sugar for

```
f = function( parameters )
   ...
end
```

which more properly shows what is happening when Lua executes the definition. When Lua executes a function definition, the function is *closed*. Different function instances, *closures*, can have different running environment, if they refer to parent block local variables. For example

```
f = function( a, b )
   return function() a+b end
end
```

creates a function which returns value of $a + b$ at the time when function f was called. As introduced in section 1.4, functions can return multiple values using $return$ with multiple parameters separated with comma. As previous example suggested, Lua also supports anonymous and non-global functions. Variable number of arguments is supported by defaulting all arguments to nil. Lua also supports proper tail recursion, in other words recursion at the end of the function re-uses the same stack space for the call, so that tail recursion can have unlimited depth.

2 C/C++ Integration

2.1 Overview

Lua can be used as a language for operating system scripting, but the main domain of application in Lua usage has always been *enhancing* applications with scripting. This section describes the basics of Lua and C/C++ integration.

In the 2.2 subsection, the core Lua library C language interface is introduced. Even with existing high level C++ Lua wrappers, it is useful to have some understanding of underlying mechanics of Lua library to take better advantage of the language. Also if you encounter any problems the understanding of low level core interface becomes useful. In the same section some of the problems with 'raw' Lua interface are covered as well. This serves as motivation for more high level interface to the library.

The subsection 2.3 introduces C++ techniques available to make Lua usage more convenient and less error prone. Techniques include for example usage of function overloading and templates for deducing types automatically and providing more type safety in general.

The last subsection describes usage of C++ wrapper for Lua which utilizes the techniques introduced in 2.3. The wrapper provides functionality for very easy to use method reflection to make functions usable by Lua scripts. When reflecting a function to Lua the user doesn't need to explicitly specify function's parameters, their types or return value of the function. In addition, the wrapper provides similar functionality other way around too, in other words to make calls to Lua functions from C++.

2.2 Lua C Interface

One of the many attractive points of Lua is that it is easy to get started using it. Lua implementation is fully reentrant, which means that it has no global variables. Lua library state is contained in *lua_State*, which is initialized and deinitalized by calling *lua_open* and *lua_close*, respectively. Script compilation can be done as simply as by calling *lua_dofile* with Lua state and name of the script file as parameters. So far so good. Unfortunately, things get a bit more complicated when we start calling functions defined in Lua scripts, and even more complicated when we want Lua scripts to be able to call our functions.

Calling functions defined in Lua scripts is heavily based on manipulation of Lua *virtual stack*. Lua virtual stack is used to pass values to and from C. Assume for example that the programmer wants to call Lua script function which is an entry in a Lua hash table, a very common scenario in Lua usage as tables are used frequently as rougly corresponding to objects in C++, so table functions correspond to C++ methods as well. To achieve described relatively simple operation, the programmer needs to:

1. Push table reference to Lua virtual stack.
2. Push hash table key, that is the function name, to Lua virtual stack with *lua_pushstring*.
3. Call Lua C interface *lua_gettable* function with correct Lua virtual stack index, which specifies where to find the table where the hash table entry, defined by previously pushed key, is located. In this case correct index is -2, which means that the table to be used in this operation is the second element from the top of the stack backwards.
4. Push any function parameters, say only number x, to Lua virtual stack with *lua_pushnumber*. Arguments are passed in stack in direct order, which means that the first argument to the function needs to be pushed to the virtual stack first as well.
5. Execute the actual script function by calling Lua *lua_call* with correct number of parameters that the function should expect. Note that function arguments are not type checked, you need to check the argument types yourself if you want to be sure that programmer didn't pass incorrect arguments to the Lua virtual stack, say by calling lua_pushstring when he should have called lua_pushnumber.
6. After the function has been executed by Lua interpreter, the parameters returned by Lua function are returned on Lua virtual stack.

7. Parameters are not type checked either, so the programmer needs to verify types manually, for example by calling *lua_isnumber* for each returned number parameter.

8. Programmer needs to retrieve the parameters by calling for example *lua_tonumber* with correct index to retrieve actual return value.

9. At the end, programmer needs to clear returned values from the Lua virtual stack by calling *lua_pop(n)*.

Or, the same thing as C code:

```
lua_getref( luastate, table );
lua_pushstring( luastate, "myfunc" );
if ( !lua_istable(luastate,-2) )
  error( "invalid table reference" );
lua_gettable( luastate, -2 );
lua_pushnumber( luastate, x );
if ( !lua_istable(luastate,-2) )
  error( "cannot call non-function" );
lua_call( luastate, 1, 1 );
if ( !lua_isnumber(luastate,-1) )
  error( "function myfunc didnt return number" );
float returnval = lua_tonumber( luastate, -1 );
lua_pop( luastate, 1 );
```

Mechanics of usage are similar when we want to be able to allow Lua scripts to call our C functions. C functions need to have declaration of

```
int scriptableCFunction( lua_State* luastate );
```

to be able to be called from Lua script. Arguments are again received in direct order, and return values are left to the top of the stack in direct order as well. Actual return value integer represents number of return values left to the top of the stack.

Needless to say, programming Lua this way is tedious and highly error prone. For example the programmer might accidentally call a wrong table by using off-by-one index which is an easy mistake to make. Even Lua 5.0 Reference Manual suggest using macros or some other high level mechanism for actual Lua library usage.

2.3 C++ Techniques for Lua Integration

Luckily, C++ provides mechanisms to make Lua usage effortless in actual application. Initialization and script compilation is already straightforward with Lua C API, but stack management was the problem in the examples of the previous subsection. Lua virtual stack management can be made considerably easier to use with features of modern C++.

Obvious first place to start when abstracting Lua virtual stack usage is the Lua hash table type, which is in heavy use especially when using Lua tables as C++ object counterparts. In the C++ wrapper which was used in the Appendix A source code example this abstraction was done with C++ class *LuaTable*. LuaTable provided basic table operations like *setString*, *setNumber*, and so on, to avoid burdening application programmer with Lua low level stack management. This already makes Lua table usage close to trivial. So instead of writing

```
lua_getref( luastate, table );
lua_pushstring( luastate, "mystring" );
if ( !lua_istable(luastate,-2) )
  error( "invalid table" );
lua_gettable( luastate, -2 );
if ( !lua_isstring(luastate,-1) )
  error( "variable is not string" );
char mystring[ENOUGH_SPACE];
strcpy( mystring, lua_tostring(luastate,-1) );
lua_pop( luastate, 2 );
```

the programmer can simply write

```
String mystring = table.getString("mystring");
```

without any consern of Lua virtual stack management details.

More difficult target for abstraction is the functions calls. If we look at the calls made from C++ to Lua scripts, we see that only things which can differ by different calls are:

1. Name of the Lua script function to be called.
2. Number of parameters given to the function.
3. Type of each parameter.
4. Optional return value and it's type.

Name based calling of a Lua script function is trivial to abstract. It can simply be done in similar fashion as the table and string usage example showed. For example *table.call("myfunc");* could be a simple call to Lua script function without parameters or return values.

Number of function arguments and their types are a bit more difficult task. The argument types can be abstracted by providing overloaded functions to push values to Lua virtual stack, so that the programmer does not need to know actual types pushed as Lua function parameters. Number of arguments can be abstracted by providing overloaded versions of the previously introduced *call* abstraction for different number of parameters, say, from zero to 10 parameters. Now we can type *table.call("myfunc",123,"hello");* without virtual stack management issues.

Receiving function calls from Lua is the hardest issue to tackle. Ideally, we'd like to write a C++ function as is and then just register it to Lua by making a

simple call like *luastate->registerFunction("myfunc", myfunc);*. This ideal can be achieved by making the registerFunction as template function which has the function type F as template parameter. Clearly, at the time registerFunction is called, the compiler knows the declaration of the function parameter. This type can be used to form a new template function with the same calling convention as regular Lua C function. This function in turn knows F, and can forward the function call to the correct function.

Appendix A contains source (by the author) of Lua wrapper which shows how the techniques introduced in this text work in practice.

References

[Ier03] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, *Lua 5.0 Reference Manual*. Technical Report MCC-14/03, PUC-Rio, 2003.

[Sco00] Scott, M., *Programming Language Pragmatics*. Morgan Kauffman, 2000.