Lua – The Next Generation

Moonshine

As a scripting language, Lua is fairly unique. It has a strong following from two niche markets: Linux users, and computer game programmers. What is it about this language that appeals? What is so special? Steven Goodwin investigates.

BY STEVEN GOODWIN

From a linguistics point of view, Lua could be considered fairly ordinary!

It has all the necessary features to make it a usable language, but lacks any strong selling points in the vein of Perl’s regular expressions, or C’s speed of execution. Its strengths do not lie with individual parts of the language, but how it functions as a whole, and how well it connects with the outside world.

We shall be looking at Lua as a customization and configuration tool, showing how the end user can customize particular software (using elinks as an example), and what steps must be taken in order to implement such functionality in their own applications.

The Whole Of The Moon

Lua began in 1993 as a language for Brazilian academics at Tecgraf, in the Pontifical Catholic University of Rio de Janeiro (PUC-Rio). Its purpose was to provide a simple method of extending applications through a basic procedural language, traditional datatypes, and a virtual machine. All these features have remained a fundamental part of Lua to the present day, with the release of version 5.0.2 (for minor bug fixes) on March 17, 2004. Its uptake has been as steady (some would say slow) as its update cycle; just 12 public releases in its entire 11 year history. However, as a compensation, there is little chance of code breaking overnight, and every release so far has been incredibly stable.

The basic structure of Lua (pronounced LOO-ah, and whose name means moon in Portuguese) is that of a language interpreter (the program, lua) which generates and runs byte code, a set of optional basic libraries (for input and output, maths, and so on, all written in standard C), and a compiler (luac) to generate byte code offline. Because of the highly standard nature of the Lua code base, it works on almost any platform that sports a C compiler. The Lua version clocks in at just over 100K, with the combined set of default libraries taking a further 72K. Many embedded projects have taken advantage of its small size (and its ability to build anywhere), as have several games companies, such as Criterion Studios, LucasArts (Grim Fandango), and BioWare (with MDK2 and Baldur’s Gate). For a longer list of uses visit [1].

Harvest Moon

As a language, it has all the features you’d expect from a functional scripting language, including the obligatory online manual, available at [2]. First off, we have the data types. These may not be plentiful (see Table 1: Lua Types), but they are capable of satisfying a programmers usual desires. The variables themselves follow the ideals of a loosely-typed language, and so can hold any type it pleases, at any particular time. Attempting to use undefined variables causes them to have the type nil, which can cause some operations (for example, string concatenation) to fail. The Lua concept of a ‘number’ is equivalent to the C type double. However, performance junkies (using version 5) can change this to float (or even int) and recompile Lua. Just add

```lua
#define LUA_NUMBER float
```

before #includeing lua.h. Anyone using version 4, or earlier, will need to manually modify the code a little further.

There is also a range of flow control statements, that fit the following:

- do block end
- while exp do block end
- repeat block until exp
- if exp then block end
- if exp then block else otherblock end
- if exp then block elseif exp then otherblock end [and so on…]
- for var = start,end,[step] do block end
- for var in iterator do block end

Listing 1: hello.lua

A comment, about a wry comment, commenting about programmer originality!

```lua
function hello()
    write(“Hello, Lua!”)
end

hello()
```

Table 1: Lua Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Identifier</th>
<th>Name for lua_intype etc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil</td>
<td>LUA_TNIL</td>
<td>nil</td>
</tr>
<tr>
<td>Number</td>
<td>LUA_TNUMBER</td>
<td>number</td>
</tr>
<tr>
<td>Bool</td>
<td>LUA_TBOOLEAN</td>
<td>boolean</td>
</tr>
<tr>
<td>String</td>
<td>LUA_TSTRING</td>
<td>string</td>
</tr>
<tr>
<td>Table</td>
<td>LUA_TTABLE</td>
<td>table</td>
</tr>
<tr>
<td>Function</td>
<td>LUA_TFUNCTION</td>
<td>cfunction</td>
</tr>
<tr>
<td>Userdata</td>
<td>LUA_TUSERDATA</td>
<td>userdata</td>
</tr>
</tbody>
</table>

Notes

- The identifier can be used to translate the constant into a string inside C, using lua_typename(lua_State *, int type)
- lua_tonumber accepts both numbers (123) and numerical strings (“123”)
- lua_toboolean returns 0 for false and nil, and 1 for everything else

Image: Moonshine

Image: Lua – The Next Generation

Image: Lua
Nothing unusual there, although C programmers should be aware that the ‘end’ value is also evaluated in the for loop. Note that all statements use the word end as a block terminator, instead of the more traditional brace. This simplicity is a very obvious attempt to coax non-programmers into the world of scripting. The preference of words over symbols is also apparent when you realize the operators !, && and || have been replaced with their alphabetic counterparts, not, and, and or.

The syntax also removes some of the restrictions found in other languages. For example, two (or more) parameters can be returned from functions, without any problem or esoteric magic.

Finally, Lua contains local and anonymous functions. Taking these in order, a local function is one that can only be called from within the function it is declared in. This is uncommon to C programmers, but fairly familiar to everybody else. Secondly, anonymous functions mean we can embed an entire function in the place we would place a routine with:

```lua
function onetwo()
    return 1,2
end
one,two = onetwo()
```

Table 2: Hooks and Crannies

<table>
<thead>
<tr>
<th>Hook function</th>
<th>Called when...</th>
<th>Should return...</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto_url_hook(url, current_url)</td>
<td>A URL is entered in the “Go to URL” dialog</td>
<td>A new URL, or nil to cancel</td>
<td></td>
</tr>
<tr>
<td>follow_url_hook(url)</td>
<td>A URL has been selected</td>
<td>A new URL, or nil to cancel</td>
<td></td>
</tr>
<tr>
<td>pre_format_html_hook(url, html)</td>
<td>Document has been downloaded</td>
<td>Modified string, or nil if no changes were made</td>
<td>Can remove adverts/junk from badly designed web pages</td>
</tr>
<tr>
<td>lua_console_hook(string)</td>
<td>Something is entered into the Lua console (type, in elinks)</td>
<td>A command (run, eval, goto_url, or nil) followed by an appropriate argument (the program to run, Lua code to evaluate, a URL to go to, or nil, respectively)</td>
<td></td>
</tr>
<tr>
<td>quit_hook()</td>
<td>elinks is about to exit</td>
<td>Nothing</td>
<td>For tidying up resources</td>
</tr>
</tbody>
</table>

Dancing in the Moonlight

Hooks are a method whereby a program (in this case, elinks) calls a special function every time it is about to do something important. This ‘something important’ might be when jumping to a URL, or downloading a page of HTML from the server. Under normal, non-hooked, circumstances, this special function will do nothing! Nada. Zip. It will simply return control back to the main program and let it jump to the URL it was going to jump to in the first place.

However, when a hook is placed into this special function, control is passed out of the main program into the hook function. At this point, the hook function has control of the data and can re-write the URL, for instance. As these hooks are programmed from a Lua program – our Lua program – we can re-write them according to our personal preferences.

For example, I may wish to visit one of my own web sites, www.BlueDust.com, by typing bd. I could do this by creating a simple hook into the ‘jump to URL’ routine with:

```lua
if (url == “bd”) then
    return “www.bluedust.com”
end
```

Lo and behold, instant customization!

The demonstration program for this article, elinks, has a number of different hooks, and these are given in Table 2: Hooks and Crannies. We can use some of these to customize the application.

As well as elinks calling our script, it is possible for our script to call elinks through callbacks. This lets us retrieve information, such as the title page, that isn’t passed through as a parameter. This is useful with key shortcuts as what parameters should be passed in to them? URLs? Bookmark lists? This is where the callback functions come into play.

These callbacks are specific functions that elinks has decided that we (as a script) may use. It allows us to call them if they were part of the Lua script itself. They include functions such as current_url and current_title. A list is

Listing 2: goto_url_hook

```lua
function goto_url_hook (url, current_url)
    if url == “tiny” then
    return url
end
```

Listing 3: bind_key function

```lua
bind_key (“main”, “Ctrl-T”,
    function ()
        return “goto_url” -
    end )
```
our code is contained within functions, nothing should appear to happen on screen.

The first of our functions will include the code for goto_url_hook. As previously mentioned, this gets called whenever the user hits ‘g’ to change web page. It is therefore a simple matter to write Listing 2.

It really is as easy as that! Reload elinks and visit your least unfavorite web site. Then press ‘g’, followed by the keyword tiny, and return. If you’re like me and chose Google as your test site, you’ll be taken to a web page at tinyurl.com, as shown in figure 1.

We can see the new url given as http://tinyurl.com/161, which can then be copied, pasted, emailed, and generally misused. If we knew how to add shortcuts to elinks we could save ourselves four keystrokes. Those of you who read Table 3: Odds and Sods earlier, already know about the function called bind_key. Yes! It is the right one, so we can add the code as in Listing 3.

This example demonstrates the usefulness of anonymous functions, and the ease by which two values can be returned from a function. In this case, the command goto_url, and the URL parameter for that command.

To add some final polish we will eradicate the duplicate URL information by writing our hooks.lua file as Listing 4.

As you can see, Lua makes it very easy for the end user to add functionality to a piece of software. You need nothing more than the methods provided here, and a little imagination, to add a whole host of other functionality. Generally speaking, adding flexibility for the end user means complexity for the programmer. With Lua, this is not the case! Let’s look at why…

Moonraker

In any extensible system like this, there are three basic components. The initialization (and de-initialization), the communication in (where the script talks to our C program), and the communication out (where C talks to the script).

Listing 4: hooks.lua file

```lua
function get_tiny(url)
end

function goto_url_hook (url, current_url)
    if url == "tiny" then
        return get_tiny(current_url)
    end
    return url
end

bind_key ("main", "Ctrl-T", function () return "goto_url", get_tiny( current_url() ) end )
```

All three areas are very simple, and can use the basic templates presented here. This simplicity has kept the Lua interpreter small, and encouraged programmers to use it for configuration.

Let us start at the beginning,

Listing 5: factorial routine

```lua
int compute(int n) { return n<2?1:n*compute(n-1); }

int c_factorial(lua_State *pLua) {
    intparams = lua_gettop(pLua);
    int n, result;
    int answers = 0;
    for(n=1; n<=params; n++) {
        if (lua_isnumber(pLua, n)) {
            result = compute(lua_tonumber(pLua, n) );
            lua_pushnumber(pLua, result);
            answers++;
        } else {
            // Error!
            break;
        }
    }
    return answers;
}
```

Moonraker

In any extensible system like this, there are three basic components. The initialization (and de-initialization), the communication in (where the script talks to our C program), and the communication out (where C talks to the script).
The first thing to note is that I’m running Lua 5. This can lead to minor compatibility issues, since `lua_open` takes a stack size parameter under version 4. Version 4 users will note that the header file must be changed to `lua40/lua.h`. These are two of the backwardly-incompatible changes made to version 5. When compiling, we must link with the Lua library:

```
$ gcc -llua mycode.c
```

If you are using Lua on its own, you will need some form of input and output. This is not included as standard, since most applications provide their own I/O. To allow Lua code access to its standard libraries, include the following:

```
lua_baselibopen(pLua);
lua_iolibopen(pLua);
lua_strlibopen(pLua);
lua_mathlibopen(pLua);
```

Which libraries you need is, obviously, up to you. However, including any of them requires linkage with the `lualib` library. Use of the maths library requires C’s own math library (which is not included as standard), to make the compile line look more akin to this:

```
$ gcc -llua -llualib -lm mycode.c
```

`pLua` holds the state of the entire Lua system. Since Lua is re-entrant, we can call `lua_open` as many times as we want, and neither state will conflict with the other.

This allows us to use Lua as part of a threaded system. Whenever interacting with Lua, we must use this pointer, which is conventionally labeled `L`, although I am using `pLua` to increase its visibility in the examples.

Having got the Lua state, we can feed Lua code to it, and the inbuilt interpreter will process it as normal.

```
lua_dostring(pLua, "number=12345");
// other code
lua_dostring(pLua, "print(number)");
```

### Table 3: Odds and Sods

<table>
<thead>
<tr>
<th>Function name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enable_systems_functions()</code></td>
<td>Allows certain functions (e.g. <code>openfile</code>) to be used. See Box 1.</td>
</tr>
<tr>
<td><code>current_url()</code></td>
<td>Retrieves the URL of the current elinks page.</td>
</tr>
<tr>
<td><code>current_link()</code></td>
<td>Retrieves the link that is currently selected (or <code>nil</code> if none).</td>
</tr>
<tr>
<td><code>current_title()</code></td>
<td>Retrieves the title from the page.</td>
</tr>
<tr>
<td><code>current_document()</code></td>
<td>Retrieves the HTML page, as a string.</td>
</tr>
<tr>
<td><code>current_document_formatted(width)</code></td>
<td>Retrieves the HTML page, formatted to the optional width.</td>
</tr>
<tr>
<td><code>pipe_read(command)</code></td>
<td>Executes the command given and reads data.</td>
</tr>
<tr>
<td><code>execute(command)</code></td>
<td>Executes the command given under a shell (using <code>sh -c</code>), like C's own math library (which is not included as standard), to make the compile line look more akin to this,</td>
</tr>
<tr>
<td><code>bind_key(keymap, keystroke, function)</code></td>
<td>Executes the function whenever the keystroke is made. Should return a command and parameter, like <code>lua_console_hook</code></td>
</tr>
</tbody>
</table>

---

**Don’t miss another DVD – upgrade your subscription today!**

You can upgrade your subscription from Standard (magazine only) to DVD (magazine and DVD) at any time – you’ll start getting the DVD with the next issue to be mailed.

For fastest service, use our secure online form: [http://www.linux-magazine.com/Upgrade](http://www.linux-magazine.com/Upgrade)
As long as we keep hold of pLua, any variables set will remain in the Lua state. Any time we call a function (like print), Lua will evaluate it with whatever functions have been declared, and return any results to the state held in pLua.

You can probably see how easy it would be to build your own interpreter and debugger with just this simple function, and you’d be right! However, such a function has already been written for us. It’s called dofile. It executes the code within the file, as if it were running from the command line: that is, executing only those statements that are global. However, unlike the command line, once the file has been run, the state remains in pLua. These variables, along with any declared functions, can now be accessed by the C code. Or another Lua file, loaded with luaL_dofile. Or code evaluated with luaL_dostring.

```c
int result;
result = luaL_dofile(pLua, “config.lua”);
```

In this case result returns the effect of the last chunk, which depends on whether the script executed successfully or not.

**Moonlight on Vermont**

Let us now write something useful in our config.lua script that calls a function in our C program. This would act like the current_url function in elinks, for example.

**Listing 6: Calling Lua**

// The function name is a global symbol: we must use that instead of a string containing the function name
lua_getglobal(plua, “swap_greeting”);
// Our first parameter
lua_pushstring(plua, “Steev”);
// Our second parameter
lua_pushstring(plua, “Hello”);
// The call itself
lua_call(plua, 2/*number of input args*/, 2/*number of result args*/);
// Retrieve results into Lua variables
lua_setglobal(plua, “result2”);
lua_setglobal(plua, “result1”);

To do this, we need to register one of our C functions with Lua. This ties the two languages together. We give it the name that Lua should use, followed by the name of our actual function in C.

```lua
lua_register(plua, “factorial”, c_factorial);
```

From here, we can get Lua to take over. It will organize all the function calling, parameter passing, and the returning of the results. All we need to do is retrieve the parameters in the correct order, and pass the correct result back.

Since Lua supports multiple return parameters (and arguments of any arbitrary type), we can not use any particular C prototype to manage it. Instead, all parameters are pushed on (and pulled off) a stack. We can then query the stack to tell us how many items are on it, and what type these items are. The stack, like variables, can hold any supported data type. It is therefore up to us (as the C programmer) to correctly request the appropriate type when removing data from the stack. This has the added bonus of making sure that every C function has the same signature, or prototype, when used with Lua.

If our factorial routine could take an arbitrary number of integers, and return the factorial of each one, we’d retrieve the number of arguments, and compute the result for each one – provided it was numeric. Our code would look something like Listing 5.

Note the stack indices are counted from 1 (not 0), and that we need to return the number of parameters that are pushed on the return stack. That’s all there is to writing your own functions. The types may vary, so lua_isnumber might become lua_isbool (see Table 1), but the principles are the same.

**Clouds across the Moon**

Calling a Lua function from C is no more difficult, once you know the pattern. It works on the same principle as before where you push data onto the stack. In this case we must place the function name on first, followed by each of the arguments in order. Since the types vary between C and Lua, you need to use the correct function to push the appropriate type onto the stack. See Listing 6.

This would have the same effect of Lua calling its own function thus,

```lua
result1, result2 = swap_greeting(“Steev”, “Hello”);
```

Note the order of the parameters is reversed to cope with their removal from the stack in first in-last out, order.

If we had wanted the results of this function in the C code, we would read the data from the stack ourselves, and then have to explicitly remove it. Like so,

```c
pResult2 = lua_tostring(pLua, -1);
pResult1 = lua_tostring(pLua, -2);
lua_pop(L, 2);
```

Notice the reverse order here, too, and the use of negative stack indices.

**Shepherd Moons**

As you can see in these few pages, we have been able to learn the basics of a new language, upgrade a web browser, and find a way to supplement our own projects with dynamic configuration scripts! This has all been possible because of the power and flexibility of Lua. I’m sure you’ll have your own ideas for projects: perhaps you’ll add the ability to email pages (or links) directly from Mutt, or open a secondary browser (for brain-dead sites that require Mozilla), or remove adverts from particular web sites, or… something else… Go on! Have a play!

**INFO**

[2] Lua documentation: http://www.lua.org/manual/5.0/
[3] Lua forums: http://archive.neotonic.com/archive/lua-

**THE AUTHOR**

When builders go down the pub they talk about football. Presumably therefore, when footballers go down the pub they talk about builders!

When Steven Goodwin goes down the pub he doesn’t talk about football, or builders. He talks about computers. Constantly…