Assignment 9

Embedding Scripting Language in script!

All Parts Due: 11:59 PM, Monday, April 11, 2016

In this assignment, you will learn some of the benefits of embedding a programming language inside an existing (and otherwise fairly ordinary) program to make it more extensible. You’ll also see the “glue” that is needed to bridge from one language to another, and see two instances of interposition. (In short, we will add scripting to script, interposing within an interposer!)

Figure 1: Normal interaction with a command-line tool.

Figure 2: How script performs interposition to create a logfile.

Setup

Use Subversion to copy the hw9 directory in the usual way. (If you have forgotten how, see the course Wiki for details.) As usual, you may work in pairs and we recommend you do.

Overview of Your Task

In last week’s lab, you used the Unix script program to log your session. The basic principles of this program are shown in Figures 1, 2, and 3. The idea of sliding code between two things that were previously coupled together is known as interposition.

The script command is usually used to record a log of a terminal session, but we can suppose that there might be potential to do other things besides the simple logging logic shown in Figure 3. Unfortunately, as it stands, the only option for people with interesting ideas for things to do is to modify the C source for script and recompile it, which is cumbersome. In other words, as it stands, the script program is not very extensible.

If you missed using the script program, do so now so that you’re familiar with it.
Embedding an Interpreter

One way to add extensibility to a program is to embed a programming language inside it. Figure 3 shows the general idea—the C code calls into code in the embedded language, which can call back out to the C code. Here, the new code merely replicates the original behavior of the program, but now we have the flexibility to make it do something different. Typically, the languages most amenable to embedding are “scripting languages.”¹

Scripting languages that support embedding in C programs include Python, Perl, Ruby, JavaScript, Lua, and numerous others. But some are more amenable to the task than others. Python, Perl, and Ruby are large (for example, the Python interpreter is over 100 times the size of the script program),² are typically not written with ease of embedding foremost, and are not actually very fast interpreters (which may not matter for our example,

¹It is, however, possible to embed other languages. Fabrice Bellard’s tcc can dynamically compile and load C code, with very fast compile times and surprisingly good code generation.

²Admittedly, the script program is a very simple program.
Figure 5: Core logic of the script program in Lua.

but might in some other program). Modern JavaScript implementations are quite fast (thanks to JIT compilation), but often even larger (e.g., 200 times the size of the script program).

In contrast, Lua is compact (e.g., my copy of the Lua interpreter is a little over 200 KB in size), relatively speedy for an non-JITed interpreter, and has ease of embedding as a design goal. Unlike some other languages that claim to be designed for ease of embedding, Lua is small without being hard to program or lacking in expressiveness.³

Figure 5 shows actual Lua code to implement the core logic of the script program. This code is also at the bottom of lscript.lua in the assignment directory.

**YOUR TASK, PART A: UNDERSTANDING LUA EMBEDDING**

In the files example.c and example.lua you’ll find some sample code showing how to interface with C. In this part, we’ll study and extend that code. None of this code will be used directly in the lscript program, but the experience you gain doing this part will give you familiarity with the concepts involved in working with an embedded Lua interpreter that will make doing Part B much more straightforward.

Proceed as follows:

1. Run make to build the example, run it, and then read example.c and example.lua carefully so that you understand how to call Lua from C, and C from Lua. You can look up any of the Lua library functions the code uses in the Lua reference manual (which you can find in the resources section of the assignment wiki page).

2. The built-in Lua math table contains a number of useful mathematical functions, but several useful functions from the C math library are not included, including tgamma, lgamma, erf, erfc, cbrt, log1p, scalbn, and frexp. But, as you have seen, we can wrap any C function and provide it to Lua. We can also write entries to Lua tables from C.

³ In contrast, some languages that target embedding lack sufficient features to avoid clumsiness, the language Tcl being one of the worst offenders!
• **Note:** All of these math functions are documented in the system which are documented in the system man pages (e.g., `man tgamma` for information about `tgamma`).

  (a) In `example.c`, write a Lua wrapper for `tgamma`, and add it to the math table from within the C code. (If the user does not provide an argument, `tgamma` should behave similarly to the standard Lua functions. If the user provides too many arguments, they should be silently ignored. This behavior requires very little effort or special code from you—especially if you use the `luaL_check*` functions instead of `lua_to*` functions. See the reference manual for details...)

  (b) In `example.lua`, write a function, `fact`, that calls `tgamma` (on the basis that \( n! = \Gamma(n + 1) \)). Show that it works by extending the `testit` function to print out a factorial.

  (c) Also add `scalbn` (which takes two numbers and returns one), and `remquo` (which takes two numbers and returns two).

  (d) In `example.lua`, write a function in `calc` that takes four arguments of different types (C or Lua), performs some calculation using all of your newly provided functions, and returns two numbers as its result. One of the arguments to `calc` must be a string. What `calc` does and how meaningful it is up to you. It can print out information about what it calculates, if you like.

  (e) In `example.c`, call `calc`, giving it a binary string that includes at least two zero bytes (the Lua function `lua_pushlstring` makes it easy to pass arbitrary strings), and print the results. You should make sure the Lua code sees the entire string, not just the string up to the first zero byte.

  (f) **Bonus (Not Required):** Also support and test `lgamma`, `erf`, `erfc`, `cbrt`, and `log1p` in `example.c`. All these functions have the same signature as `tgamma` (they take and return one double). You should add them in a way that avoids lots of duplicated identical code in `example.c`.

---

**Your Task, Part B: Embed Lua in Script, Move Key Functionality to Lua**

In this part, you’ll modify the `script` program to embed Lua. The `script` program traces its ancestry back to 3.0 BSD (1979), although it has been modified in various ways over time; our version is based on the Mac OS X version, which in turn is based on FreeBSD’s version.⁴ I have modified the code slightly to better suit our needs in this assignment, and to compile on Linux. Our version of `script` is called `lscript`, and is in `lscript.c`.

Even though `lscript.c` is a short C program, I do not expect you to understand the totality of how `lscript` works (because we will be making a few targeted changes and mostly deleting code). Nevertheless, we can observe a few things about the general flavor code. In particular, it has almost no comments, and makes liberal use of global variables. Your additions should have comments if/when appropriate, but you may follow the pattern of the code and use global variables freely. The code is written in C89, which

---

⁴ The Linux version of `script` is based on the same codebase, but has been modified heavily to add various bells and whistles.
Table 1: C Functions from lscript to provide to Lua and their Lua names.

<table>
<thead>
<tr>
<th>C Function</th>
<th>Lua Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>writetosubprocess</td>
<td>writeToSubprocess</td>
</tr>
<tr>
<td>writetouser</td>
<td>writeToUser</td>
</tr>
<tr>
<td>setperiod</td>
<td>setPeriod</td>
</tr>
<tr>
<td>done</td>
<td>exit</td>
</tr>
<tr>
<td>isechoing</td>
<td>terminalIsEchoing</td>
</tr>
</tbody>
</table>

has a few strange limitations (hence the way variables are declared first at the beginning of functions). Your additions may use C99 features, but you may not switch the code over to C++.

Proceed as follows (run svn commit between each part in case you make a mistake):

1. Add the following to lscript.c
   (a) Includes for using Lua (but put them after the includes already in the file).
   (b) A global variable for the Lua state
   (c) Code at the start of main to initialize the Lua interpreter, load the standard Lua libraries, and load & execute lscript.lua.
   (d) Code at the end of done to shut down the Lua interpreter.

   If you do this part correctly, now when you run the lscript program, it should print “Lua code loaded!” to standard error when the program starts.

   • **NOTE:** You’re not calling any of the Lua functions in lscript.lua yet.

2. Wrap and export the C functions shown in Table 1 so that they’re available to Lua. Note that when wrapping writetosubprocess and writetouser you should use lua_checklstring not lua_checkstring. See the Assign9HintsAndTips page for why.

3. Modify lscript.c so that instead of calling writetosubprocess and writetouser, it instead appropriately calls the Lua functions userInputEvent and subprocessOutputEvent (when passing a string to these functions, use lua_pushlstring; in both cases the length of the string is in the cc variable). Also, change the C function periodic so that it calls the Lua function periodic (in addition to flushing the log).

   • **NOTE:** You can ignore all code related to writetolog, we’ll handle that later.

   • **NOTE:** You’re not calling the init and terminate functions in lscript.lua yet.

4. Test lscript to make sure it still works.

5. Experiment with lscript.lua by adding a call to either writeToSubprocess or writeToUser in the Lua function periodic, and invoke ./lscript -t 3 to make sure that your added action happens every three seconds. (Comment out your added code when you’re done.)
6. Delete all code related to opening, writing to, and closing the log file from the C code, including informational messages—an easy way to do so is to delete the declaration for the variable fscript and then search for lines referring to that variable and remove them. Make sure the program still “works” (i.e., except for writing the log).

7. Store the filename argument to lscript in a Lua global variable called logname. Once this variable is set, call the Lua function init. Also, at the start of the done function, call the Lua function terminate.

8. Test the lscript program, logging should now be restored, but with Lua managing the task of writing to the log.

9. Delete all the option flag variables from lscript.c, as well as the code that sets them. Instead of the switch statement used in option processing, put all the passed options into the options table (a global variable created by the Lua code). Note that when an option has a value, we want to store that value in the table entry corresponding to the option, otherwise we will store the boolean value true as the value for that option. Thus, we want

```
./lscript -q -t 5
```

to cause the equivalent of

```
options.q = true;
options.t = "5";
```

It may help to know that for options that have values, C’s getopt function makes the global variable optarg point to a string containing that value, otherwise it holds NULL.

**Your Task, Part C: Use What You Made**

Modify lscript.lua to do something “interesting”. Possible options include

- Simulating an old-style modem by storing I/O events and using periodic to perform them more slowly.
- Write a tool that records or replays a session (using timing information).
- Writing something that annoys people in some way.
- Something else...

(Something utterly trivial gets half credit. Something awesome gets extra credit.)

---

5 You can, if you prefer, transfer even more of the option processing into Lua. In the sample solution, the option-specification string that gets passed into the getopt is set from lua code (before init is run), allowing the lua code to add new options for the program.
You may add additional helper functions in C if you really need them, but remember that Lua already has quite extensive libraries, so there is a good chance that whatever you need is already there.

**NOTE:** You can have fun having your friends (possibly unwittingly) try your lscript tool, but you may not do anything to capture passwords from unsuspecting users. We'll consider that a violation of the honor code, even if you can claim “No harm was done”, or “I didn’t realize!” If necessary, stop and think about how a session with a friend might play out before they sit down and start typing. If you screw up, you must self report (as always).