Experimental Meta-Programming for Lua

Max Rottenkolber <max@mr.gy>

Friday, 18 December 2015

Working on *Snabb Switch* (`https://github.com/snabbco/snabbswitch`) has made me appreciate the *Lua* (`http://www.lua.org/`) programming language quite a bit, and a *JIT compiled* (`https://en.wikipedia.org/wiki/Just-in-time_compilation`) implementation of the language, called *LuaJIT* (`http://luajit.org/`), specifically. Lua is a neat little programming language. Simple enough that you can learn to use it effectively in a matter of days, yet surprisingly powerful: higher-order functions with language support for multiple return values and an ubiquitous structured data type—a hybrid between a sparse array, a hash table and prototype objects[1]—make Lua a breeze to hack in. LuaJIT is a ridiculously pragmatic programming language implementation: because of Lua's simple semantics, its "just-in-time" compiler is able to generate insanely fast native code. Additionally, LuaJIT provides a sophisticated *C FFI* (`http://luajit.org/ext_ffi.html`) that not only makes it trivial to call C functions, but also allows you to make very effective use of C data structures within Lua programs. In fact, LuaJIT makes using C structures so seamless, you can even define Lua methods on them. I have been thoroughly impressed by how Snabb Switch hackers can write elegant high-level code in one situation, and craft super-efficient hot paths, where the budget is literally measured in "instructions per packet", in another, all in the same language!

At the same time, I am a *big* fan of Common Lisp. When I am in the position to dictate a programming language, I will choose Common Lisp, and usually, when I have to use something other than Common Lisp, I start to miss it quickly. Interestingly, Lua has been able to elude my dissent, and LuaJIT has revealed itself as an especially good fit for the Snabb Switch project. Still, as a lisper I sometimes wonder, what if I could write macros? What if Lua's syntax was based on *S-expressions*?[2] As for LuaJIT's FFI extension, with power comes responsibility. It is not very hard to make LuaJIT produce segmentation faults when dealing with foreign objects. Null-pointer dereference, "use after free" bugs, its all fair game when using the LuaJIT FFI extension. Maybe, the safety

hazards could be isolated using macros. I thought about that for a while, and remembered how I had enjoyed using *Parenscript* (`https://common-lisp.net/project/parenscript/`) to meta-program *Javascript* in the past. Parenscript is impossible to use unless you know your way around both Javascript *and* Common Lisp, but still it enhanced my Javascript experience notably. I figured, why not try something similar for Lua?

- 1. "Prototype" as in prototype based object orientation.
- 2. Also, what if there wasn't as much of a expression/statement dichotomy? But I digress. :-)

S-expressions and Macros for Lua

S-Lua (`https://github.com/eugeneia/s-lua`) is an experiment in wrapping Lua in S-expressions, adding macros and exploiting the hack for meta programming profit. The S-Lua abstraction is designed to be as thin as possible, only changing Lua's syntax while leaving the semantics of the language untouched. Additionally, it adds a macro expansion time in between reading and loading code. S-Lua is aware of three types of data: strings, numbers, and tables.[1] It is implemented in Lua, and its macros are implemented as Lua functions that return abstract syntax trees represented by nested tables. It has no notion of a *cons cell* and its symbols are just unquoted strings. Below is an illustration of S-Lua's syntax.

S-Lua	**Read as**	**Compiles to**
`(a b c)`	`{a, b, c}`	`a(b, c)`
`(a (b))`	`{a, {b}}`	`a(b())`

```
% luajit -i s.lua
> prin1(read"(print 42)")
{
  "print",
  42,
}
> =compile(read"(print 42)")
print(42)
> sLua"(print 42)"
42
```

This does not get us far. We can now call functions, but other Lua constructs have dedicated syntax and cannot be expressed this way. These built-in Lua constructs have to be implemented as S-Lua built-ins as well, and S-Lua implements those as close to their Lua counterparts as possible.

S-Lua	**Compiles to**
`(return` *a b*`)`	`return` *a*`,` *b*
`(vector` *a b*`)`	`{`*a*`,` *b*`}`
`(=` `(`*a b*`)` *c d*`)`	*a*`,` *b* `=` *c*`,` *d*
`(local` `(`*a b*`)` *c d*`)`	`local` *a*`,` *b* `=` *c*`,` *d*
`(do` *a b*`)`	`do` *a*`;` *b* `end`
`(for` `(_` *a*`)` *b c d*`)`	`for _,` *a* `in` *b* `do` *c*`;` *d* `end`
`(function` *name* `(`*a b*`)` *c d*`)`	`function` *name* `(`*a*`,` *b*`)` *c*`;` *d* `end`

Some S-Lua built-ins in a nutshell.

Fair enough, the only thing missing now are macros. The other half of S-Lua is comprised of a Common Lisp inspired back-quoting system and a facility to install macros. The back-quote syntax is just a list comprehension tool—or table comprehension in our case. It lets us conveniently build abstract syntax trees by splicing symbols, strings (quoted symbols) and tables into other tables. The back-quoting syntax is really just a sugar coating at the reader level that expands to `quote`, `unquote` and `splice` forms at read-time. The former is implemented as a built-in, and the latter two only have meaning within invocations of `quote`.

S-Lua	Expands to	Evaluates to
`‘`*a*	`(quote` *a*`)`	`"`*a*`"`
`‘(`*a* `,`*b*`)`	`(quote (`*a* `(unquote` *b*`))`	`{"`*a*`",` *b*`}`
`‘(`*a* `,@(`*b* *c*`))`	`(quote (`*a* `(splice (`*b* *c*`))))`	`{"`*a*`",` *b*`,` *c*`}`

S-Lua	Compiles to
`(defmacro ...)`	Similar to `function` but installs function as a macro

Now we have everything required to write our first macro. In Lisps you can typically find the `let` form, which is a construct for lexically binding variables around other forms. Below is a simplistic version of `let` implemented in S-Lua. Instead of using Lisp-typical list processing functions such as `car`, `cdr` or even `mapcar`, our `let` uses idiomatic Lua table mangling to achieve its purpose.

```
(defmacro let (bindings ...)
  (local (vars vals) (vector) (vector))
  (for (_ b) (ipairs bindings)
    (= (vars[#vars+1] vals[#vals+1]) b[1] b[2]))
  (return ‘(do (local ,vars ,@vals) ,...)))
```

Basic `let` macro in S-Lua.

```
> loadFile("let.sl")
> princ(macroExpand(unpack(read"(let ((x 42)) (print x))")))
(do (local (x) 42) (print x))
> =compile(read"(let ((x 42)) (print x))")
do local x = 42; print(x) end
> sLua"(let ((x 42)) (print x))"
42
```

Fair enough, that works. And to my surprise, the code for `let` is not even that ugly. It makes successfully use of array indexing and Lua's unary # operator, which is lucky since S-Lua has no concept of these

language constructs. From S-Lua's point of view, these are just symbols.[2] Equally lucky, it also manages to correctly use Lua's special "vararg" (. . .) symbol. Neither the macro nor the code it produces are especially "lispy", but rather close to idiomatic Lua. I think that is pretty neat.

- 1. Bug: S-Lua can not really read quoted strings, yet.
- 2. For dynamic indexing or use of the # operator it would require the respective S-Lua built-ins.

A "Practical" Use Case

Snabb Switch has a core function `packet.free` used to deallocate foreign C structures that contain packet data. I recently wrote a *bug* (`https://github.com/SnabbCo/snabbswitch/pull/664`) in which I dereferenced a pointer to a packet that was already freed, leading to a segmentation fault. That made me think about "defensive programming", and if we could somehow retain at least memory safety in this scenario. I imagined a macro that would free a packet in a *place* and then discard its pointer, to eliminate the hazard of dereferencing it another time.

```
(defmacro pfree (place)
  (return `(do (packet.free ,place)
               (= ,place nil))))
```

`pfree` frees the packet in *place*.

```
> =compile(read"(pfree p)")
do packet.free(p); p = nil end
```

The `pfree` macro might be silly, and there are other solutions to this problem such as boxing the packet pointers, but it achieves something that can not be replicated in plain Lua: it does not operate on a value, but instead expects its argument to be a *place*—a symbol denoting a variable or object slot from which a value can be read and assigned to.

But is it any good?

S-Lua has some advantages over Parenscript: because it is implemented in its target language, using it does not require semantic understand-

ing of yet another programming language. While Parenscript forces you to semantically understand three languages—Javascript, Common Lisp and Parenscript itself—S-Lua only requires you to understand Lua and its own semantics, which are arguably marginal. Another cool thing is that S-Lua can dynamically compile code into a running LuaJIT process using `loadstring`. I am not exactly sure what the performance implications of `loadstring` are—e.g. if the resulting code will be JIT compiled—but in theory LuaJIT should be able to optimize the code without restrictions.

The downside to S-Lua—as felt by all source-to-source compilers—is that it obfuscates compile- and run-time errors. There is no easy way to make Lua errors meaningfully correspond to S-Lua source code. Only further efforts can tell if there is a sweet spot between S-Lua trying harder to produce corresponding code and restricting its use to a minimum—punctually invoking S-Lua in a source code preprocessor could leave most Lua code undisturbed. The mental overhead introduced by S-Lua might be more than can be justified by the features provided by it. In any case, it was a fun experiment!