Luma - Lpeg-based Lua macros

Fabio Mascarenhas

December 7, 2007

Motivation

- Macros that are not bound by Lua's syntax...
- ...but clearly delimited when mixed with Lua code
- Defining a macro's grammar and the code it generates should be as easy as possible
- Lpeg's re.lua for grammar and Cosmo for code templates
- Inspired by Scheme's *define-syntax*

A Luma Macro

```
#!/usr/bin/env luma

require_for_syntax[[automaton]]

local aut = automaton [[
  init: c -> more
  more: a -> more
        d -> more
        ' ' -> more
        r -> finish
  finish: accept
]]

print(aut("cadar"))
print(aut("cad ddar"))
print(aut("caxadr"))
```

Using Luma Macros

- Macro application: *selector* `[[` *data* `]]`
- *data* is any string that contains balanced `[[`s and `]]`s
- It's up to the macro to parse and interpret it
- Expansion is repeated until there are no more macros to expand, so macros can expand to macros
- Command line script, *luma*, expands macros in files before running them
- *require_ for_ syntax* macro require's a library in expansion time, so any macros defined by the library are available to the code that follows it
- Automaton's syntax is far from Lua's
- Similarity to Converge's *DSL blocks*

Defining *automaton* - Syntax

A Luma macro needs a *syntax*, a set extra *definitions and builders*, and a *code template*. The syntax for *automaton*:

```
local syntax = [[
  aut <- _ state+ -> build_aut
  char <- (['][{[ ]}['] / {.}) _
  rule <- (char '->' _ {name} _) -> build_rule
  state <- ( {name} _ ':' _ rule+ -> {} {'accept'?} _ ) ->
]]
```

Luma uses re.lua, and defines a few default rules that can be used by any macro: spaces and Lua comments (_), Lua names (*name*), Lua numbers (*numbers*), and Lua strings (*string*, *shortstring*, and *longstring*).

Defining *automaton* - Definitions

automaton's definitions are the capture functions, they will build a description of the automaton suitable for use by the code template.

```
local defs = {
  build_rule = function (c, n)
    return { char = c, next = n }
  end,
  build_state = function (n, rs, accept)
    local final = tostring(accept == 'accept')
    return { name = n, rules = rs, final = final }
  end,
  build_aut = function (...)
    return { init = (...).name, states = { ... },
      substr = luma.gensym(), c = luma.gensym(),
      input = luma.gensym(), rest = luma.gensym() }
  end
}
```

Defining *automaton* - Code Template

Luma uses *Cosmo* for its code templates, Cosmo templates expect a table (the first capture the grammar returns on match) and use this table to fill the template.

```
local code = [[
  (function ($input)
    local $substr = string.sub
    $states[=[
      local $name
    ]=]
    $states[=[
      $name = function ($rest)
        if #$rest == 0 then
          return $final
        end
```

continues...

Defining *automaton* - Code Template contd.

```
        local $c = $substr($rest, 1, 1)
        $rest = $substr($rest, 2, #$rest)
        $rules[==[
          if $c == '$char' then
            return $next($rest)
          end
        ]==]
        return false
      end
    ]=]
    return $init($input)
  end)]]
```

Macro hygiene is currently the macro programmer's responsibility.
Also Cosmo supports iteration over list items.

Using Lua's syntax

Defining macros that use a subset of Lua's syntax is easy, just use *Leg*'s Lua parser. If Luma detects that Leg is present it modifies the parser so macro applications are legal Lua syntax.

Example, list comprehensions:

```
x = L[[i | i <- 1,3]] -- x = {1, 2, 3}
y = L[[L[[j | j <- 1, 3]] | i <- 1, 3]]
    -- y = {{1, 2, 3}, {1, 2, 3}, {1, 2, 3}}
z = L[[tostring(automaton[[init: c -> more
                           more: a -> more
                                 d -> more
                                 r -> finish
                           finish: accept]](line))
      | line <- io.lines()]]
    -- tries to recognize each line from stdin with
    -- the automaton and collect results in z
```

List comprehensions - Syntax

The list comprehension macro imports the *Exp* rule from Leg as *exp*:

```
local syntax = [[
  comp <- (_ {exp} _ '|'  _ (numfor / genfor) _)
    -> build_comp
  numfor <-  ({name} _ {~ '<-' -> '=' ~} _
    {exp _ ',' _ exp _ (',' _ exp)?}) -> concat
  genfor <- ({name} _ (',' _ name)* _ {~ '<-' -> 'in' ~}
    _ {exp _ (',' _ exp)*}) -> concat
]]
```

The builders are straightforward, *concat* concatenates all captures separated by whitespace, *build_ comp* creates a table with the expression, the *for* part and a gensym for the final list.

List comprehensions - Code

The code is very straightforward, as the expression and for parts are already in the correct Lua syntax:

```
local code = [[
  (function ()
    local $list = {}
    for $for_part do
      $list[#$list + 1] = $exp
    end
    return $list
  end)()
]]
```

Inlining definitions

Luma defines a *meta* macro that lets you execute arbitray code at expansion time. This lets you define macros in the same file they are used:

```
meta[[
local function fact(n)
  n = tonumber(n)
  if n <= 1 then return 1 else return n * fact(n-1) end
end
luma.define("fact", "{number}", fact)
]]

print(fact[[3]])
```

If a macro is very simple you can use a function that returns its expansion instead of a Cosmo template.

Final Remarks and Issues

- Currently Luma has several other examples: class definitions, try/catch/finally, inc, match/with using Lpeg, using, Python-like *from module import symbol*, nor...
- Most macros defined in less than 50 lines of code, the simplest in less than 10 lines, including templates
- But no satisfactory error reporting (just a generic parse error if a macro doesn't match)
- gensym is also brittle, uses ___luma_sym_*n* convention, could lead to conflicts, and of course very easy to forget
- selector `[[ data ]]` syntax has the problem of not allowing an unbalanced `]]` anywhere inside a macro application, needs a way to escape this string

I'm open to questions and suggestions!