A PEG-based Macro System for Lua

Fabio Mascarenhas[1], **Sérgio Medeiros**[1]

[1]Departamento de Informática – PUC-Rio
Av. Marquês de São Vicente, 225 – Rio de Janeiro – RJ – Brazil

mascarenhas@acm.org, smedeiros@inf.puc-rio.br

Abstract. *However not available in many languages, macros are an essential feature in languages such as Lisp and Scheme. We will present Luma, a macro system for the Lua language that uses Parsing Expression Grammars (PEG) as a pattern-matching mechanism.*

Resumo. *Embora não estejam disponíveis em muitas linguagens, macros são uma característica fundamental de linguagens como Lisp e Scheme. Apresentaremos Luma, um sistema de macros para a linguagem Lua que utiliza gramáticas de parsing de expressões para realizar o casamento de padrões.*

1. Introduction

Macros are an extension mechanism very popular in languages such as Lisp and Scheme, which define many of their basic constructs through macros. Albeit macros are heavily used by Scheme and Lisp users, macro systems are not present in many programing languages, and those languages that support macros, like C, only do it in a limited way.

However the Lua language does not have a native macro system, currently there are several options that enable the use of macros in Lua. We will present a new macro system for Lua, called Luma, which is based on the idea of Parsing Expression Grammars.

This paper is organized in the following way: section 2 presents an overview of Luma; section 3 shows an example of a macro using Luma; section 4 presents a brief comparison between Luma and other macro systems; and section 5 draws some conclusions.

2. An Overview of Luma

Luma is a macro system for the Lua language that is heavily inspired by Scheme's `define-syntax`/`syntax-rules` system [N. I. Adams et al. 1998]. The Scheme macro system uses pattern matching to analyze the syntax of a macro, and template substitution to build the code that the macro expands to. This is a powerful yet simple system that builds on top of Scheme's structural regularity (the use of S-expressions for all code).

Luma also separates the expansion process in pattern matching and template substitution phases, but as Lua source is unstructured text the pattern and template languages have to work with text. Luma uses *LPeg* [LPeg 2007] for pattern matching and *Cosmo* [Cosmo 2007] for templates.

LPeg is a Lua implementation of the *Parsing Expression Grammars* (PEG) formalism [Ford 2004], an EBNF-like formalism that can define a parser for any unambiguous language. PEG is different from other grammatical formalisms in that it can easily be used

to define both the lexical and the structural (syntactical) levels of a language. Cosmo is a template language not unlike template languages used for web development, and supports both simple textual substitution and iteration.

A Luma macro then consists of a *pattern*, a *template* and optional definitions typically used to produce an environment suitable for Cosmo from the LPeg output. A template can also be a function that Luma calls with the captures produced by LPeg and which returns a template for expansion. When you define a macro you also need to supply a *selector*, which Luma will use to identify applications of that macro in the code it is expanding. A selector can be any valid Lua identifier.

Macroexpansion is recursive: if you tell Luma to expand the macros in a chunk of Lua code Luma will keep doing expansion until there are no more macro applications in the code. This means that Luma macros can expand to code that uses other macros, and even define new macros. A macro application has the form:

```
selector [[
    ... macro text ...
]]
```

where the macro text can have balanced blocks of `[[` and `]]`. If Luma has no macro registered with that selector it just leaves that application alone (in effect it expands to itself). This syntax for macro applications is not completely alien to Lua, as most macro applications will be valid Lua code (a function call with a long string), but also guarantees that all macro applications are clearly delimited when mixed with regular Lua code.

When expanding a macro Luma invokes LPeg with the macro's pattern and the supplied text, then passes the template and captures to Cosmo, and replaces the macro application with the text Cosmo returns (recursively doing macroexpansion on this text).

The next section shows a complete implementation of a Luma macro, including an example of its application.

3. Walkthrough of a Luma Macro

The macro example in this section is taken from a paper on Scheme macros [Krishnamurthi 2006]. We think it is a good example because the syntax for this macro is quite removed from normal Lua syntax, in effect the macro embeds a domain specific language inside Lua, which compiles to efficient Lua code via macroexpansion. Our macro defines *Deterministic Finite Automata*. A description of the automaton is supplied as the macro text, and it expands to a function which receives a string and tells if the automaton recognizes this string or not. The following Lua code uses the macro to define and use a simple automaton:

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

```
print(aut("cadar"))  -- prints "true"
print(aut("caxadr")) -- prints "false"
```

You execute this code by saving it to a file and calling the Luma driver script, *luma*, on it. The driver macroexpands the file and then supplies the expanded code to the Lua interpreter. The `require_for_syntax` macro requires a Lua module (in this case the module `automaton.lua`, which will define the `automaton` macro) at expansion time, so any macros defined by the module are available when expanding the rest of the code.

An automaton has one or more states and each state has zero or more transition rules and an optional tag that marks that state as a final (acceptance) state. Each transition rule is a pair of a character and a next state. The macro's syntax codifies that in a LPeg pattern (a Lua string):

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

LPeg patterns extend regular PEG with captures: curly brackets ({}) around a pattern item tell LPeg to capture the text that matched that item, and a right arrow (->) tells LPeg to pass the captures of the pattern item to the left of the arrow to the function to the right (or collect the captures in a Lua table in the case of ->{}. The functions are part of the optional set of defitions. Luma defines a few default expressions that can also be referenced in patterns: spaces and Lua comments (_), Lua identifiers (`name`), Lua numbers (`numbers`), and Lua strings (`string`). The first PEG production is always used to match the text (`aut` in the case of the pattern above).

The defitions for the `automaton` macro contain the functions referenced in the syntax:

```
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
}
```

This is regular Lua code. What these functions do is to put the pattern captures in a form suitable for use by the template. As Luma uses Cosmo this means tables and lists.

The top-level function, `build_aut`, builds the capture that is actually passed to Cosmo, so it also defines any names that the code will need to use for local variables so they won't clash with the names supplied by the user (in effect this is manual macro hygiene [Kohlbecker et al. 1986]).

The template is a straightforward tail recursive implementation of an automaton in Lua code:

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

Cosmo replaces standalone *$name* by the corresponding text in the table that Luma passes to it, and forms such as *$name*[=[*text*]=] and *$name*[==[*text*]==] by iterating over the corresponding list and using each element as an environment to expand the text (as an example, if *foo* is a list containing the elements { *bar = "1"* }, { *bar = "2"* }, and { *bar = "3"* }, *$foo[=[$bar]=]* becomes the string *123*.

Defining the macro is a matter of telling Luma the selector and components of the macro:

```
luma.define("automaton", syntax, code, defs)
```

As Luma uses LPeg as pattern language, macros that embed part of the Lua syntax (creating a localized syntactical extension) can be easily built by using a Lua parser written for LPeg, such as *Leg* [Leg 2007]. The Luma distribution has samples such as a class system, try/catch/finally exception handling, list comprehensions, Ruby-like blocks, and declarative pattern matching, all built using Luma and Leg.

4. Other Macro Systems

4.1. LuaMacro

LuaMacro [LuaMacro 2007] is a macro system based on tokens transformation that offers functionalities similar to the C preprocessor. It is very easy to write simple macros, like factorial, using LuaMacro but, as long as the macros become more complex, the macro programming becomes harder too. Because LuaMacro only see tokens, there is not a notion of the structure of a construction, thus the programmer can not programm in an abstract way, which sometimes is confusing.

Albeit this problem, it is possible to use LuaMacro to define many of the macros that you can define using Luma, like a try/catch exception handling and a Python-like syntax for Lua. It is possible to see some of these more complex examples on the LuaMacro site [LuaMacro 2007].

4.2. MetaLua

Metalua [Metalua 2007] is an extension mechanism that supports marcros. Metalua works with an Absctract Syntax Tree (AST), so we can deal with the language constructions in a more abstract and structured way.

Metalua has the concept of metalevels and has also, like Scheme, quoting and splicing operators (**+{...}** and **-{...}**, respectively), which allow to change between metalevels. When a splicing operator is not inside a quoting operator this means the code delimited by the splicing operator should be executed at compile time. Using Metalua it is possible to write macros to modify the Lua parser, adding constructs like a try/catch/finally exception handling, or just write macros as functions that will be executed at compile time.

We can define *automata* as a Metalua macro that will be called like a function, where the first argument is the initial state, the second one is a table representing the final states and each following argument is a table representing a state of the automata. The macro will be executed at compile time, and each time it is executed it will return a function that produces a new automata following the description supplied. The following code illustrates this:

```
local aut = -{automata ('init', {'finish'},
           {'init', {'c', 'more'}},
           {'more', {'a', 'more'}, {'d', 'more'}, {'r', 'finish'}},
           {'finish'} )
}

local myaut = aut ()
print (myaut ("caddr"))
print (myaut ("cadra"))

local otheraut = aut ()
print (otheraut ("cdadaaaddddr"))
```

When *aut* is called it returns a new automata and we can use this new automata to check if a string is reconized or not.

5. Conclusion

Currently, there is no native macro system for the Lua language, however, several options are emerging and have been discussed on the main mailing list for the language [lua-l 2008].

We presented Luma, a macro system inspired by the Scheme macro system and based on LPeg and Cosmo. Luma allows to extend the basic syntax of Lua, enabling the programmer to use a syntax that barely resembles Lua (for the good and for the bad).

As Luma works with patterns and templates, it provides a more abstract mechanism than LuaMacro, that works on tokens. Luma can also take benefit of Leg, a Lua parser, in a way that is somewhat similar to Metalua, that also offers a parser for the Lua language, so we can match an expression, the *else* part of an *if*, and so on.

All the macro systems discussed here, however, are not mature enough yet, and a lot of development need to be done. Finally, albeit all this macro proposals that are coming in the last years, there is still a reasonable amount of doubts if the Lua language really needs a macro system or not.

References

Cosmo (2007). Cosmo: a safe template engine. Available at http://cosmo.luaforge.net. Visited on January 2008.

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA. ACM.

Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B. (1986). Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA. ACM.

Krishnamurthi, S. (2006). Educational pearl: Automata via macros. *Journal of Functional Programming*, 16(3):253–267.

Leg (2007). Leg: LPeg-powered Lua 5.1 grammar. Available at http://leg.luaforge.net. Visited on January 2008.

LPeg (2007). LPeg: Parsing Expression Grammars For Lua. Available at http://www.inf.puc-rio.br/ roberto/lpeg.html. Visited on January 2008.

lua-l (2008). Lua mailing list. Archive available at http://lua-users.org/lists/lua-l. Visited on Januray 2008.

LuaMacro (2007). LuaMacro: A Macro Facility for Lua using Token Filters. Available at http://luamacro.luaforge.net. Visited on January 2008.

Metalua (2007). Metalua: Lisp macro's power brought to Lua. Available at http://metalua.luaforge.net. Visited on Januray 2008.

N. I. Adams, I., Bartley, D. H., Brooks, G., Dybvig, R. K., Friedman, D. P., Halstead, R., Hanson, C., Haynes, C. T., Kohlbecker, E., Oxley, D., Pitman, K. M., Rozas, G. J., G. L. Steele, J., Sussman, G. J., Wand, M., and Abelson, H. (1998). Revised5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76.