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.

Resumo.

1. Introduction

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 [**?**]. 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* [**?**] for pattern matching and *Cosmo* [**?**] for templates.

LPEG is a Lua implementation of the *Parsing Expression Grammars* (PEG) formalism [**?**], 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. The 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's 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 [**?**]. 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 an 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 extends regular PEG with captures: curly brackets ({}) around an 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 [**?**]).

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* [**?**]. 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. Conclusion