Evitar que las funciones en Lua «choquen» con las macros de LaTeX en LuaTeX

Juan Manuel Macías

2 de diciembre de 2019

LuaTeX ofrece muchas y variadas maneras de manipular el contenido textual en distintos estadios del proceso de compilación, en lo que va del código fuente al objeto tipográfico resultante. Hay vías más triviales y otras de mayor complejidad, pero todas parten de esa característica esencial de que goza la avanzada versión de TeX de puentear las primitivas del cajista binario mediante *scrips* en Lua, ya que incluye un intérprete para este lenguaje liviano y multiusos.

Una forma limpia y rápida de manipulación, pero con un poderoso alcance en cuanto a resultados, es mediante el añadido (un injerto, más bien) de funciones en Lua que incluyan sustituciones de cadenas de texto. Este tipo de funciones deben anclarse o registrarse en el *callback* de LuaTeX `process_input_buffer` que, como su nombre anuncia, actúa sobre el `input` en el momento que éste es analizado para su compilación. Ya dimos algunos ejemplos aquí, pero ahora nos detendremos en algunas particularidades más y cierto problema colateral.

El *callback* `process_input_buffer`, en efecto, remite a un momento muy tempranero del proceso. Tan tempranero que las macros de LaTeX que tengamos dispersas por el código fuente todavía no se han expandido, y por tanto pueden verse fácilmente afectadas por nuestras funciones de sustitución de caracteres. Por ejemplo: imaginemos que queremos sustituir todas las apariciones de la letra «b» minúscula por la secuencia `\textbf{b}` (o sea, que toda ocurrencia de la «b» quede en negrita en el *output*). En cuanto el análisis llegue a alguna macro de LaTeX que contenga esta letra (como un simple `\bigskip`), la compilación se detendrá por un error insalvable. Y no es para menos, porque antes ya se habría operado allí la sustitución, y lo que tendría que compilar sería entonces un ininteligible `\\textbf{b}igskip`. Así que, de entrada, y antes de echar mano de este recurso tan útil, conviene valorar antes los pros y los contras que acarreará el hacerlo. En ocasiones, nada hay que temer, como en este ejemplo que puse al final de mi entrada sobre el problema del punto alto griego, donde se ajustaba la altura del glifo mediante un `string.gsub` de Lua. Es improbable que una macro de LaTeX incluya un punto alto griego. En otros contextos, cuando lo que esperamos es una sustitución lineal de un carácter por otro, será mejor considerar crear al vuelo una propiedad OpenType de reemplazo, mediante la función nativa de LuaTeX `font.handlers.otf.add.feature`[1].

[1] Esto puede ser muy productivo, por ejemplo, para transliteraciones directas entre alfabetos, como el latino y el gótico.

Habrá casos, sin embargo, donde nos toparemos sin remedio con el problema de frente. ¿Cómo solucionarlo? Bien, en un hilo de `tex.stackexchange.com` un usuario aportó una función que intentaba evitar el desaguisado mediante un condicional. Aunque me parece muy ingeniosa la idea del condicional, la función (en parte porque era poco menos que un esbozo) tenía un par de problemas de peso:

1. Impide que se «toquen» las macros simples de LaTeX, pero no las que tienen argumentos obligatorios u opcionales, de modo que puede seguir habiendo obstáculos en la compilación.
2. Tampoco permite cribar aquellos argumentos donde sí quisiésemos que se operase una sustitución (por ejemplo, en un `\textit{...}`.

Por tanto, he probado a intentar «mejorar» esa función, añadiendo más condicionales, afinando un poco más y ampliando las variables. De todo eso, extraigo aquí un mínimo ejemplo operativo.

Supongamos que en nuestro documento de LaTeX queremos que se sustituyan al compilar todos los casos de «p» y «m» por las cadenas «Pili» y «Mili», respectivamente. Por supuesto, no queremos que se sustituyan dentro de las macros ni de sus argumentos, pero sí en el interior de algunos argumentos, en este caso sólo dos: `\textit{...}` y `\section{...}`.

Para empezar, tenemos que separar la estructura de una macro de LaTeX en seis variables, y a cada una de ellas le asignaremos luego una captura de expresiones regulares, o patrones[2] en la jerga de Lua, opcionales u obligatorios en su aparición:

a la barra invertida

b una secuencia de caracteres

c la llave izquierda

d secuencia de caracteres que puede contener espacios

e la llave derecha

f secuencia de caracteres que puede incluir un corchete izquierdo y otro derecho

Definimos ahora una función simple para la sustitución de los caracteres «p» y «m»[3]:

[2] No son las expresiones regulares, dicho sea de paso, el punto fuerte de Lua, pues no sigue aquí el estándar POSIX. Con un sistema de expresiones regulares más completo, probablemente no serían necesarias tantas vueltas y el asunto se resolvería de una manera más quirúrgica.

[3] Téngase en cuenta que todo código Lua que insertemos en nuestro preámbulo debe encerrarse en un entorno `luacode`, del paquete del mismo nombre.

```lua
function sustitucion (texto)
   texto = string.gsub (texto, "p", "Pili")
   texto = string.gsub (texto, "m", "Mili")
   return texto
end
```

Como dije, con esta función sola no vamos a ningún lado, porque en cuanto una macro de LaTeX tenga una «p» o una «m» estaremos acabados.

Así que, a continuación, la función que hace el resto, y que voy comentando entre medias del código:

```lua
function sustitucion_buena ( texto )
   -- Aquí definimos antes una variable local con las 6 capturas a
   -- buscar; la parte que sustituye es una función anónima
   local x =
   ↪  texto:gsub('(\\?)([%a%@%s]+)([{]?)([%s%a%@]*)([}]?)([%[%]%s%a%@%w]?)',
   ↪  function (a,b,c,d,e,f)
                           -- Este primer condicional nos permite tocar el
                           ↪  interior de textit y section
                           if (a~="" and b=="textit" or b=="section")  then
                              -- Defino esta variable, para sustituir dentro
                              ↪  de textit y section:
                              y = string.gsub (d, "[%s%a%@]",   sustitucion)
                              -- y lo concatenamos todo:
               return a .. b .. c .. y .. e .. f
                           end
                           -- El segundo condicional es para no tocar
                           -- el resto de macros de LaTeX y que no nos
                           -- dé error en la compilación
                           if (a~="" or c~="" or d~="" or e~="" or f~="")
                           ↪  then
                           return a .. b .. c .. d .. e .. f
                           end
               -- Si no se dan las dos condiciones previas, entonces
               -- ya lo que tenemos es la cadena simple a sustituir
            b = string.gsub (b, "[%a%@%s]",   sustitucion)
            return b
         end)
      return x
end
```

No nos queda más que definir el comando de LaTeX que cargue la función:

```
\newcommand\sust{\directlua{luatexbase.add_to_callback
    ( "process_input_buffer" , sustitucion_buena , "sustitucion_buena" )}}
```

Y probarla. Para ver bien los resultados, nada mejor que un entorno `verbatim` (y el resultado de la compilación en fig. 1):

```
\begin{document}
  \sust
     \begin{verbatim}
        p y m
       \textit{p y m}
       \section{p y m}
       Pero esto no lo sustituye:
       \textbf{p y m}
       \cualquiermacro{p y m}[p y m]
     \end{verbatim}
\end{document}
```

```
 Pili y Mili

\textit{Pili y Mili}

\section{Pili y Mili}

Pero esto no lo sustituye:

\textbf{p y m}

\cualquiermacro[p y m]
```

Figura 1: Compilando nuestra sustitución de cadenas