AskLua: añadiendo ayuda interactiva a otros módulos

Julio Manuel Fernández-Díaz

Profesor Titular, Departamento de Física de la Universidad de Oviedo

Febrero 2010

Índice general

AskLua: añadiendo ayuda interactiva a otros módulos

Julio Manuel Fernández-Díaz

Profesor Titular, Departamento de Física de la Universidad de Oviedo (España), Febrero de 2010

Resumen:

Se presenta `AskLua` un sistema de gestión de ayuda para Lua: en línea desde el intérprete interactivo, en formato `html` y en formato impreso.

El módulo `ask`, proporcionado por `AskLua`, es poco intrusivo ocupando memoria que puede ser liberada por el usuario en cualquier momento si no desea seguir con la ayuda en línea.

El sistema está bastante bien integrado, de tal manera que se puede añadir fácilmente ayuda para un módulo ya existente, incluso de tipo binario.

1 Introducción

Cuando aprendemos una nueva materia es conveniente disponer de información lo más completa posible sobre el tema. Muchas veces esto se logra con un experto (profesor). Sin embargo, no siempre tenemos acceso a uno, por lo que hay que ser autodidacta. En esas ocasiones un buen sistema de ayuda, lo más interactivo posible, es de gran ayuda.

Tanto Python como Matlab, por poner dos ejemplos, tienen ayuda interactiva en línea. Esto permite a los principiantes un cómodo acceso durante las primeras fases de aprendizaje del sistema.

Obviamente una herramienta como los lenguajes citados tendrá más uso si se facilita el acceso a la documentación. Y eso le interesa mucho al diseñador de un módulo concreto.

Lua tiene un enfoque casi contrario a los dos lenguajes citados. Por tanto, es prácticamente una quimera que pueda incluir un sistema de ayuda en el estándar.

Lua no impone *políticas*, pero es muy extensible. Esto nos permite incluir fácilmente un sistema (módulo) de ayuda, que se carga cuando se desea.

La Programación literaria, muy orientada al programador, no me convence como mecanismo, pues hay que aprender otro *pseudolenguaje* y realmente sirve para otra cosa.

El sistema de documentación que acompaña a `stdlua` con `@param`, `@returns`, etc., tampoco es de mi agrado, pues consiste en comentarios al código. Esto está bien para revisar y estudiar código Lua, pero conseguir un sistema de documentación en línea en este caso es relativamente complejo. Además la documentación está también orientada al programador y no tanto al usuario de algún módulo (por ejemplo, un ingeniero que quiere usar Numlua).

Un sistema muy similar al que viene en `stdlua`, aunque más avanzado es Luadoc, que permite generar documentación en formato `html` a partir también de comentarios en los módulos. El formato final es bastante amigable para consultar la documentación en un navegador, pero no sirve para consultas en línea, que es lo que se deseaba inicialmente.

En cualquier caso, tanto el sistema de `stdlua` como el de Luadoc pueden adaptarse fácilmente a lo que se propone más abajo: basta con integrar los comentarios en las correspondientes tablas de ayuda.

Cuando estaba acabando esta documentación he descubierto en LuaRocks un módulo denominado LuaHelp (con fecha de la primera versión de 13 de diciembre de 2009). Al principio pensé que mi trabajo había sido en vano pues ya estaba hecho el módulo que yo deseaba. (De todos modos siempre se aprenden cosas nuevas al programar.) He de decir que se me pasó su anuncio en la lista de correo de lua.

Una vez analizado el tema con más profundidad, me he dado cuenta de las diferencias (aunque LuaHelp se podría adaptar de muchas maneras evidentemente): la documentación que yo propongo va incluida en el propio módulo, mientras que la otra se incluye en ficheros auxiliares situados en un camino de búsqueda concreto. Además el método aquí presentado permite consultar por separado partes concretas de la ayuda: cómo se usa una función dentro de un módulo, ejemplos de uso, la lista de funciones de un módulo, etc. Son dos enfoques distintos con sus ventajas e inconvenientes (que el usuario puede comprobar por sí mismo).

La única pega es que me ha hecho cambiar el nombre de mi módulo. Ahora se denomina `ask` (y yo hubiera preferido `help`). De esa manera se pueden utilizar ambos a la vez. De todos modos, si se desea renombrar `ask` y llamarlo `help` sólo renombrando `ask.lua` a `help.lua` es suficiente. En ese caso es interesante cambiar los avisos que un módulo da al cargarse (véase el apartado 4.3 Cómo sabemos que un módulo tiene ayuda).

Otra posibilidad puntual es cargar el módulo `ask` referenciándolo con variable de nombre `help` de la manera siguiente:

```
local help = require‘ask‘
```

Et voilà. Ya podemos usar tanto `ask` como `help` para usar el presente módulo (a partir de ese momento no podremos usar LuaHelp evidentemente).

2 El problema

Tiene dos aspectos:

1. Por un lado, para el *programador* de un módulo al que se quiere añadir ayuda. Deberá ser fácil de implementar, con las mínimas modificaciones a su módulo.

2. Por otro, para los *usuarios* del módulo. La consulta de la documentación deberá ser lo más completa posible pero de fácil manejo.

En el primer aspecto, nos interesa como programadores que sea fácil de añadir la documentación, lo más completa posible. También que sea fácil darle uno de los tres formatos siguientes:

- en línea en Lua interactivo,
- en línea en un navegador que requiere evidentemente formato `html` (es preferible este formato estándar que otros existentes),
- en formato impreso.

Además, al programador le interesa muchísimo tener que introducir la documentación una sola vez y que un sistema automático genere los diferentes formatos. Esto evita muchos errores y si hay que hacer algún cambio lo deberá hacer en un solo lugar.

Desde el punto de vista del usuario, éste querrá que sea:

- *fácil* la consulta, proporcionando sólo la información que se pida en cada momento, ni más ni menos;
- lo *más completa* posible, cubriendo muchos aspectos, incluso colaterales a veces;
- *relacional*, o sea que nos permita ir de un lugar a otro de la documentación siempre que existan relaciones entre diferentes partes de la misma.

Tras darle vueltas a la cuestión, hacer diversas probaturas y versiones del módulo de ayuda, al final he llegado al sistema que se explica en el apartado siguiente.

3 La solución

La información de ayuda en Lua sólo (bueno más o menos) puede ir en tablas, con strings, o en ficheros (como en LuaHelp). Me ha parecido que el manejo de ficheros sería bastante engorroso pues deben estar accesibles y deberían estar posiblemente donde el módulo propiamente dicho. Habría que buscar en los *path* de Lua y podría haber más de un fichero con el nombre que buscamos y no corresponder la ayuda a lo que se desea. Por tanto he preferido la otra opción.

Nuestro módulo de ayuda `ask` debe poder acceder a esas tablas (si existen y sino avisar de que no hay disponible información sobre un determinado módulo) y procesar la ayuda como se le pida.

Vamos a describir el módulo `ask`.

3.1 Dónde se almacena la información

Lo primero es *dónde* se almacena la ayuda relativa a un módulo. En Lua es evidente que debe ir en una tabla, con posibles subtablas, y al final texto (strings). Además, la ayuda de un módulo (digamos `mininum`, el ejemplo que se adjunta con `ask`), debe estar asociada al mismo de alguna manera: Lua debe conocer la ayuda al igual que conoce las funciones del módulo (sino no podría "ayudarnos").

La ayuda debe estar compartimentada. La idea final consiste en incluir en el módulo una tabla `_H` con campos y también subtablas. He aquí la estructura:

```
_H = {
  _CHARSET = "iso-8985-15",   -- otras veces "utf-8"
  _Name    = [[Nombre del módulo]],

  _basic   = [[Ayuda básica]],
  _usage   = [[Ayuda de cómo se utiliza]],
  _more    = [[Información complementaria]],
  _seealso = [[Enlaces a más información interna o externa]],
  _example = [[Un ejemplo de utilización]],
  _version = [[Versión, nombre del autor y otras cuestiones]],
  _notes   = [[Notas, como pueden ser las de Copyright, etc.]]
}
```

(Nota: fíjense en la `N` mayúscula de `_Name`.)

`_CHARSET` indica qué codificación se ha usado en los textos de ayuda. Para ayuda generada en formato `html` el sistema usa directamente el valor indicado en este campo.

Para ayuda en línea, `ask` detecta la codificación de caracteres del sistema operativo para convertir la ayuda del conjunto de caracteres indicado en `_CHARSET` al del sistema (y así visualizarse correctamente en pantalla). De momento este sistema sólo está implementado con UTF-8 e iso-8859: si `ask` no detecta UTF-8 entonces piensa que el sistema operativo trabaja con

`iso-8859`.

Esta tabla se incluye en cualquier punto del fichero `módulo.lua` (aunque lo normal es que vaya cerca del principio). Ninguno de los campos es obligatorio. Digamos que si no se presenta un ejemplo el campo `_example` no se especifica.

Como vemos, para que exista la mínima interferencia con otras variables y funciones del módulo le he puesto delante del nombre "_". Eso evitará colisiones de nombres: es raro utilizar nombres de ese tipo (excepto para variables locales). En cualquier caso el usuario de `ask` puede cambiar en el código fuente del mismo `_H` por otro nombre con un editor de textos (ya que se proporciona el código fuente) y tenerlo en cuenta en su propio módulo.

Cada una de las funciones del módulo (v.g. `about`, `base` y `doc` en el módulo `ask`) debe tener su ayuda asociada de la manera (para la primera de las funciones):

```
_H.about = {
  _basic   = [[Una descripción somera del propósito de la función]],
  _usage   = [[Una descripción completa de la manera de invocar
              la función, con sus argumentos y sus retornos]],
  _more    = [[Información complementaria]],
  _seealso = [[Enlaces a más información interna o externa]],
  _example = [[Un ejemplo de utilización]],
}
```

En algunos casos se tiene tablas de funciones. Si por ejemplo `tabfun` es una tabla con funciones, digamos `f1` y `f2`, deberíamos poner:

```
_H.tabfun = { ... }

_H.tabfun.f1 = {
  _basic   = [[ ... ]],
  _usage   = [[ ... ]],
  _example = [[ ... ]],
}

_H.tabfun.f2 = { ... }
```

Obviamente si no existe algún tipo de información pues no se especifica (como `_more` y `_seealso` en la `f1` anterior).

Inicialmente el formato del texto que va entre `[[ ]]` era texto puro. Con eso para la ayuda en línea es suficiente. Sin embargo era mi deseo que la información también se pudiera imprimir y entonces algo de formato más avanzado es mejor. Eso me llevó al formato markdown, que además tiene una implementación en Lua puro markdown.lua realizada por Niklas Frykholm. (Nota: la versión que aparece en luaforge.net está anticuada, pero la que aparece en LuaRocks es correcta.) Realmente, este documento ha sido preparado usando markdown.

Ese formato no es muy intrusivo. Esto es importante porque no deseaba hacer un filtrado de la ayuda en línea, sino que quería que apareciera tal cual está en los strings de ayuda.

3.2 Cómo se accede a la información

Bueno, ya tenemos la información en *su* sitio (tablas de Lua). Cuando se carga el módulo también se cargan esas tablas que están accesibles desde ese momento (evidentemente ocupan espacio; luego, en 5 Cómo usar la ayuda interactivamente veremos cómo liberarlo si se desea).

Ahora debemos hablar de *cómo* se accede a la información. Para ello `ask` tiene tres funciones: `about`, `base` y `doc`.

| about |

Ésta fue la primera en ser diseñada: proporciona la información en línea. Hay que indicarle *sobre qué* queremos ayuda y *cuál* de las diversas partes de la información deseamos (por ejemplo `basic`, para la información básica). Al final la función `about` se invocaría por ejemplo:

```
ask.about"/mininum.root^basic"
```

siendo `mininum` el nombre del módulo, `root` el nombre de una de las funciones dentro del mismo, y `basic` el tipo de información que deseamos (luego explico la aparición de `"/"` al principio del nombre). Como vemos separamos con un acento circunflejo `"^"` sobre qué se desea información de qué tipo de información queremos.

En otro ejemplo tendríamos:

```
ask.about"/nombre.tabfun.f1^usage"
```

o sea, le pedimos ayuda de uso para la función `f1` dentro de la tabla `tabfun` del módulo `nombre`.

Existe otra información importante para el usuario que no hace falta que la introduzca el diseñador del módulo: la lista de funciones dentro del módulo. Esa información la obtiene `ask` directamente y se presenta en pantalla mediante:

```
ask.about"/mininum^list"
```

También se puede pedir de una vez toda la información que exista sobre algo mediante:

```
ask.about"/mininum.root^all"
```

Como vemos, el usuario en este formato básico de invocación de `about` tiene que teclear bastante. Lo siguiente que hice fue admitir abreviaturas para el tipo de información pedida:

- `b` por `basic`, que proporciona información básica;
- `e` por `example`, que nos muestra un ejemplo;
- `l` por `list`, que nos da una lista de funciones en el módulo o en la tabla de funciones sobre la que pedimos ayuda;
- `m` por `more`, que nos da información complementaria;
- `s` por `seealso`, que nos cita enlaces a más información interna o externa,
- `u` por `usage`, que nos muestra cómo se usa el módulo o la función;
- `v` por `version`, que nos muestra información sobre la versión, nombre del autor y otras cuestiones;
- `n` por `notes`, que nos muestra notas como las de Copyright, etc.
- `a` por `all`, que proporciona toda la información anterior.

De esta manera se tendría la equivalencia:

```
ask.about"/mininum^list"  --><--  ask.about"/mininum^l"
```

Los espacios tecleados como argumento de `ask.about` se eliminan antes de la búsqueda. También se colapsan múltiples `"."` que pudieran aparecer en el string de búsqueda. Por ejemplo, tendríamos la equivalencia:

```
ask.about"/ mininum . root ^ usage"  --  ask.about"/mininum.root^u"
```

Por defecto se supone `basic` como información si no se proporciona.

Usando `nil` como argumento de `ask.about` obtenemos información básica sobre el sistema de ayuda:

```
ask.about()  --><--  ask.about("/ask^basic")
```

son equivalentes.

Hay que recordar que `ask` proporciona la información que el diseñador del módulo haya incluido en las correspondientes tablas de ayuda (excepto para `list` y para `all`, evidentemente). O sea que la información podría ser incorrecta si se introducen, por ejemplo, datos sobre la version como si fuera `more`.

| base |

El lector verá que todavía tenemos que teclear mucho. Normalmente cuando se trabaja con un módulo (pongamos Numlua) necesitamos ayuda de manera implícita del mismo y no de otro. Para eso he incluido la función `ask.base`. Esta función permite asumir implícitamente un prefijo de búsqueda de información. Por ejemplo, poniendo:

```
ask.base"mininum"
-->  Changing help basis to "mininum"
```

a partir de ese momento se añade el prefijo `"/mininum"` a la ruta de búsqueda (a veces con un punto `"."` al final si `basis` se refiere a un módulo o a una tabla de funciones dentro del módulo y deseamos información sobre una función concreta), y entonces son equivalentes:

```
ask.about"/mininum^list" --><-- ask.about"^l"
```

A ese prefijo lo he llamado `basis` (se guarda en una variable local).

Invocando `ask.base` usando como parámetro un string vacío `""` nos devuelve la `basis` actual:

```
ask.base""
-->   Help basis is "mininum"
```

Por otro lado, me ha parecido conveniente que

```
ask.base(nil)
```

establezca `"ask"` como `basis`.

El sistema de ayuda en línea ya está operativo, pero me gustaba poco tener que teclear `ask.about` cada vez que quería ayuda (aunque según se lee `ask.about"/mininum.root^usage"` es casi equivalente a la frase inglesa *ask about mininum.root usage*). La solución fue incluir el trozo de código:

```
__call = function (t, s, ...)
   if s == nil then
      about(nil)
   elseif type(s) == "string" then
      about(s)
   else
      return t[s](...)
   end
end

setmetatable(_G.ask, ask)
```

al final de `ask.lua`. La última línea nos permite utilizar directamente `ask` como variable global después de hacer `require"ask"`. El trozo previo de código permite que la función `about` sea invocada automáticamente cuando a `ask` se le pasa un string como argumento. O sea que a partir de este momento son equivalentes:

```
ask.about"/mininum.root^usage"  --  ask"root^u"
```

(con `basis = "mininum"`). Como podemos observar, la simplificación es manifiesta, y menos ya no se puede teclear al pedir ayuda (algo menos si se invoca al módulo `ask` de la manera: `h = require"ask"` pues a partir de ese momento se podría poner `h"root^u"`; sin embargo me parece que `ask` no es mucho escribir, es más descriptivo, y `h` es posible que tenga otra utilidad dentro de nuestro programa).

| doc |

Con lo anterior ya tenía el módulo con la funcionalidad deseada. Pero es una pena tener la información sólo en línea y no poder imprimirla en un formato decente. Por eso mismo he diseñado la función `doc`. La primera idea era adaptar un poco la salida de texto. Pero un poco más de gasto al escribir la ayuda, usando el formato markdown, permite una salida en formato `html` bastante conveniente. Eso fue fácil pues existe un módulo markdown.lua (que obviamente hay que tener instalado), el cual se puede invocar desde `ask`.

La versión `html` es complementaria a la versión en línea de la ayuda. Permite una visión más *global* de un módulo dado, y se puede consultar navegando entre la ayuda de las diferentes funciones. Por tanto, es bastante interesante.

Es en este punto donde se parece el sistema `ask` y Luadoc, en que ambos generan formato `html`. Sin embargo los puntos de partida son muy diferentes.

También se puede obtener en formato impreso, imprimiendo desde el navegador, pero quizá es mejor usar un programa como html2ps de Jan Kärrrman, el cual genera un formato de impresión (PS y luego podemos obtener PDF, también este último con hiperenlaces activos).

La función `ask.doc`, que ha sido rediseñada varias veces, y todavía no he quedado totalmente conforme con el resultado (aunque funciona correctamente, sin ser muy elegante), es la que genera `html`. En este caso, no se necesita información parcial de funciones dentro de un módulo. Por tanto, la misma genera un fichero `html` con toda la información correspondiente al módulo (para ello `basis` debe contener el nombre del módulo y no de una de las funciones del mismo). Por ejemplo:

```
ask.doc""
```

generaría la ayuda para `basis` (v.g., `"mininum"` anteriormente), en formato `html` con el nombre `"mininum.html"`, y:

```
ask.doc"ask"
```

generaría la ayuda para `ask`.

El fichero `html` generado por `ask.doc` contiene primero información genérica sobre el módulo (el contenido de `_H._basic`, `_H._usage`, etc.) Luego, la ayuda de todas las funciones (en orden alfabético), para finalizar con `_H._version` y `_H._notes`.

He de pedir disculpas por la programación un poco "sucia" de algunas partes del módulo. Mis otras ocupaciones no me han permitido más tiempo de momento, y como decía uno de mis profesores (León Garzón) "lo *mejor* es enemigo de lo *bueno*".

4 Cómo adaptar módulos para usar la ayuda

4.1 Por qué el formato markdown

El formato markdown es lo bastante simple como para que un programador de un módulo lo domine en una hora. No es necesario para la ayuda en línea (incluso hay algunos que pueden pensar en que es algo molesto). Yo creo que no introduce mucho *ruido* en la ayuda en línea y sin embargo permite una buena presentación en `html` (y en PDF tras otras conversiones).

Se puede consultar en la dirección indicada, pero también existe un documento PDF de un par de páginas impresas con una ayuda simple del sistema Markdown Syntax Cheat Sheet (la verdad es que hace falta poco más que esta ayuda simple para dominar markdown).

Pensando un poco más en la ayuda en formato `html` e impreso se pueden incluso introducir referencias a alguna imagen. Es evidente que en la ayuda en línea, en modo texto, es imposible visualizar una imagen, pero se puede dar su referencia en formato markdown que no es demasiado intrusiva. Sería de la manera siguiente:

```
![Texto alternativo](/path/to/image.jpg "Título opcional")
```

(En vez de formato JPG se puede usar también PNG o GIF.)

El formato markdown permite también indicar enlaces a otros documentos (internos o externos) de una manera simple. En la ayuda en línea no es relevante pero sí en formato `html`. Por ejemplo, `_seealso` puede diseñarse incluyendo enlaces de la manera:

```
internos:          [etiqueta](#referencia)

externos:          [etiqueta](http://lo.que.sea)
```

4.2 Preparando la información de ayuda

La idea general es sencilla: hacerlo de manera incremental. La documentación completa puede ser una labor larga y tediosa. Por tanto, habría que comenzar implementando `_basic` y luego `_usage`, pues son los dos aspectos primarios. Como ese suele coincidir con lo que se pone como comentario en la cabecera de una función lo que tenemos que hacer es convertir los comentarios ya escritos en parte de `_H`.

Posteriormente se pueden ir añadiendo diferentes partes que nos fueron quedando. Un ejemplo (comentado si se desea) suele ser conveniente para que el usuario pueda probarlo (incluso copiando y pegando código).

El lugar lógico de introducir esa información es delante de la función correspondiente (como se suele hacer con los comentarios descriptivos de su funcionalidad) y en el caso de módulos cerca del principio del fichero.

Si se desea preparar la ayuda para un módulo compilado tenemos que hacer lo siguiente.

Si el módulo binario es nuestro, y podemos recompilarlo a nuestro gusto, podemos cambiar el nombre de la tabla internamente si lo deseamos, y renombrar también el fichero compilado. Luego se crea un fichero fuente Lua, con la ayuda incluida, con el nombre antiguo pero extensión `.lua` que carga el fichero compilado.

Una segunda posibilidad es denominar de manera diferente el módulo compilado y el que vamos a cargar con `require`. Sea por ejemplo un módulo compilado en formato binario de nombre `mymodule.so` (o `mymodule.dll`). Crearemos un fichero Lua con nombre `mymodule1.lua` de la forma:

```
-- fichero mymodule1.lua
require "mymodule"
mymodule._H = { ... }
```

Para cargar `mymodule` debemos usar:

```
require "mymodule1"
```

y tendremos a nuestra disposición las funciones `mymodule.<function>` y también la ayuda `mymodule._H`, a partir de ese momento.

Otra tercera posibilidad es usar el mismo nombre, usando el hecho de que Lua primero busca ficheros fuente en `LUA_PATH` y luego binarios en `LUA_CPATH`. Crearíamos entonces un fichero `mymodule.lua` en el mismo lugar donde está el binario, de la manera:

```
-- fichero mymodule.lua
module(..., package.seeall)

local p = package.path
package.path = ""
package.loaded["mymodule"] = nil

require"mymodule"
package.path = p

mymodule._H = [[ayuda]]
```

De esa manera podemos seguir usando

```
require "mymodule"
```

como siempre, pero ahora con ayuda incluida.

De todos modos, este último método podría fallar si el módulo binario `mymodule.so` (o `mymodule.dll`) carga a su vez módulos en código fuente Lua pues durante el momento de su carga está desactivada la búsqueda en `package.path`.

4.3 Cómo sabemos que un módulo tiene ayuda

Al final de fichero Lua que contiene el módulo se puede incluir el siguiente trozo de código para detectar si Lua ha sido lanzado de manera interactiva:

```
-- checks if Lua calling was interactive;
-- it does not work in all cases, but it does in the normal ones
local interactive = true
if _G.arg then
   for _, v in pairs(_G.arg) do
     interactive = false
     if v == "-i" then
       interactive = true
       break
     end
   end
end
```

Este sistema de detección no siempre funciona bien (y eso permite generar la documentación `html` con una simple línea dentro del intérprete de comando del sistema operativo, como veremos en el siguiente apartado).

Sería interesante que en modo interactivo Lua crease una variable que lo indicara o al menos usara `_PROMPT` para ello: si no se expresa explicitamente `_PROMPT` está sin definir y Lua usa `">"` para ese cometido directamente. Bastaría sólo con definir `_PROMPT` en modo interactivo (`">"` si el usuario no definió otro) y dejarlo sin definir en otro caso.

Una vez detectada la interactividad del intérprete tendríamos el siguiente bloque de código:

```
if interactive then
   -- reusing interactive
   interactive = pcall(require, "ask")                          --**
   if interactive then
      io.stderr:write('Module "mininum" loaded. ')
      io.stderr:write('To obtain help invoke ask"mininum".\n')  --**
      io.stderr:write('Documentation occupies memory. ')
      io.stderr:write('For freeing it let execute:\n')
      io.stderr:write('\n    mininum._H = nil\n\n')

      ask.base"mininum"
   else
      io.stderr:write('Module "mininum" loaded. It has help\n')
      io.stderr:write('but module "ask" is not accesible.\n')   --**
      io.stderr:write('Help removed.\n')
   end
end

if not interactive then
   -- deleting _H
   _H = nil
   _G.ask = nil
   collectgarbage()
end
```

Esto libera memoria no necesaria en modo no interactivo, y además imprime en pantalla algunos mensajes en modo interactivo.

Si deseamos cambiar el nombre de `ask.lua` a otro nombre deberemos cambiar también las líneas indicadas por `--**` más arriba de manera acorde.

En otro orden de cosas, en el futuro es posible la inclusión de una utilidad externa para adaptar el formato que acompaña a Luadoc (y también el que acompaña a LuaHelp) para su uso con el módulo `ask`. Esto permite no partir de vacío para módulos cuya documentación ya estaba preparada de algún modo.

5 Cómo usar la ayuda interactivamente

Como antes hemos expuesto, un módulo que lleve ayuda deberá indicarlo al cargarse con `require` para que el usuario sepa que tiene disponible la ayuda.

El último módulo cargado que posea ayuda cambiará `basis` por el nombre del módulo. Esto deberá tenerlo presente el usuario. De todos modos, en cualquier momento éste podrá cambiar `basis` a su gusto o podrá de manera puntual usar el sistema con la ruta completa (que empieza con `"/"`).

Si se desea eliminar la ayuda de la memoria es fácil. Supongamos que tenemos el módulo `mininum` cargado con

```
require "mininum"
```

pues entonces basta con teclear:

```
mininum._H = nil   -- repetir esto con cada módulo
collectgarbage()
```

Luego, si queremos eliminar completamente el sistema de ayuda teclearemos:

```
_G.ask = nil
collectgarbage()
```

A partir de este momento ya no ocupa memoria ni la ayuda concreta del módulo `mininum` (aunque evidentemente sus funciones siguen cargadas), ni el sistema `ask`.

6 Cómo generar documentación en formato html (y PDF)

En este tipo de documentos es importante indicar el conjunto de caracteres usado en la ayuda. En estos momentos yo estoy trabajando con `iso-8859-15` y por tanto, en la ayuda se incluye el campo:

```
_H._CHARSET = "iso-8859-15"  -- "utf-8" en otros casos
```

Si se está usando `utf-8` (más corriente estos días) se debe cambiar de manera acorde. Si no se hace así algunos caracteres como "ü", "ñ", "á", saldrán incorrectamente. Se suele presentar la documentación en inglés por lo que el presente tema típicamente sólo afectaría a nombres de personas con esos caracteres.

Una manera muy sencilla de obtener la documentación en formato `html` sería la siguiente en modo interactivo:

```
$ lua
> require "mininum"
> ask.doc""
```

Esto generaría `mininum.html`. Sin embargo, debido a un pequeño problema en la detección de la interactividad (véase el siguiente apartado), esta misma documentación también se puede lograr mediante:

```
lua -e "require'mininum'; ask.doc''"
```

Una vez se tiene el fichero con formato `html` se puede visualizar con un navegador cualquiera para consultar la ayuda. La hoja de estilo `default.css` proporcionada sitúa en la parte izquierda un menú fijo con los enlaces de todo el documento, incluyendo la lista de funciones dentro del módulo.

El estilo del fichero `html` de salida es modificable, a través de la hoja de estilo `default.css` (nombre incluido dentro de `ask.lua`). Se recomienda sólo hacerlo en caso de conocimientos de CSS, pero no hace daño jugar con él.

Si existen muchas funciones en el módulo es posible que no se vea el final de la lista dentro de la pantalla. En ese caso se debe disminuir el tamaño de letra (o visualizar el documento sin estilo, que aunque es más feo, es todavía operativo). En ocasiones también tiene que disminuirse el tamaño de letra para ver los ejemplos (las líneas no dan vuelta en los elementos `html` de tipo `<pre>`, a los que se convierten los ejemplos).

Aparte de visualizar el documento `html` también se puede imprimir, ya sea con el navegador (no recomendable) o con un programa específico. Aquí se recomienda la utilización de html2ps, que convierte `html` en PS, con una hoja de estilo un poco simplificada (véase la documentación proporcionada en el enlace). En este caso, el menú de navegación no se imprime.

Posteriormente, si se desea, se puede convertir el fichero PS en PDF mediante el programa `ps2pdf` que acompaña a Ghostscript. Tiene este sistema la particularidad de que los enlaces se mantienen, pudiendo también navegarse con enlaceds en el fichero PDF.

7 Debilidades y posibles mejoras

En primer lugar, debido a que modificamos algunas tablas (internas en el módulo del que se quiere ayuda) es posible que haya algunos efectos colaterales (que yo todavía no he descubierto).

Por otro lado, cuando se carga un módulo en una variable local de la manera, por ejemplo:

```
local m = require"mininum"
```

para usar `"m."` como prefijo para las funciones de `mininum` y escribir menos, a partir de ese momento se puede acceder a la ayuda con `mininum` y con `m`, lo que no importa mucho en modo interactivo (al igual que se puede acceder a las funciones con `m.fun` y con `mininum.fun`).

Si embargo, `ask.doc"m"` genera un fichero `mininum.html` con el nombre del módulo `mininum` (y no el fichero `m.html`).

Las funciones en el módulo que vayan a tener ayuda no deben empezar por `"_"` pues `ask` supone que son parte de su sistema (no es que no se puedan definir y usar, sino que no se puede proporcionar ayuda).

Algunas veces la interactividad no funciona bien. Por ejemplo:

```
lua -e "require'mininum'" somefile.lua
```

imprime un mensaje al principio y además no borra la ayuda. O sea que cuando se ejecuta `somefile.lua` las tablas de ayuda están en memoria (lo que puede ser contraproducente).

Sin embargo no importa demasiado pues basta incluir

```
require"mininum"
```

al principio de `somefile.lua` e invocar a `lua` sin la opción `-e`, y entonces se descarga el módulo `ask` y la ayuda de `mininum` (una vez que detecta que el modo no es interactivo) antes de ejecutar el resto del programa.

En modo interactivo, en el futuro se puede hacer que haya enlaces `http` para ayuda más completa (por ejemplo para una descripción de un algoritmo o para dar un enlace a un artículo científico), o para comprobar si hay nuevas versiones de un módulo dado. Cuando apareciera uno de esos enlaces se podría lanzar un navegador cuando el usuario pidiera ayuda de ese tipo (habría que incluir un nuevo tipo, v.g., `link` equivalente en el modo de una letra a `k`).

También se pueden añadir test ejecutables (o sea pruebas en cada función del módulo; en parte podrían ser los ejemplos ejecutados). Por ejemplo, se incluirían dos campos: `_test` con código y `_testsolution` con texto. Al ejecutar el primero de ellos se debería obtener el segundo (el módulo `ask` comprobaría que son iguales). No sé si esto es o no interesante. Habría que estudiarlo más.

El sistema según se ha presentado no está cerrado ni es la solución definitiva. Puede tener muchas mejoras y así lo desea su autor, que sea un acicate para que otros programadores piensen sobre el tema y desarrollen incluso mejores sistemas (o completen éste).

8 Conclusiones

Se ha diseñado un interesante sistema para proporcionar ayuda en otros módulos. El mismo es versátil, permitiendo ayuda en línea, en formato `html` para su consulta en un navegador y en formato PS o PDF para impresión.

El sistema no es perfecto, pero sí operativo. El autor cree que puede ser la semilla de un sistema más completo. El lector avezado en Lua puede hacer las modificaciones y adaptaciones que crea convenientes para su uso personal. Realmente, `AskLua` se distribuye como dominio público. Está esperando mejoras.