A próxima geração

# Feitiço da Lua

Como linguagem de script, Lua é bastante única. Tem uma grande aceitação de dois nichos do mercado: usuários Linux e programadores de jogos para computadores. O que há de atraente nessa linguagem? O que é tão especial? Steven Goodwin investiga. POR STEVEN GOODWIN

De um ponto de vista lingüístico, a Lua pode ser considerada bem comum! Tem todas as características necessárias para torná-la uma linguagem utilizável (variáveis, funções e controle de fluxo), mas carece de alguns pontos importantes na linha das expressões regulares do Perl ou da velocidade de execução do C. Todavia, sua força não reside nas partes individuais da linguagem, mas em como funciona no geral e em como se conecta ao mundo externo.

Neste artigo, olharemos para Lua como uma ferramenta de personalização e configuração, mostrando como o usuário final pode personalizar certos softwares utilizando Lua (nós vamos usar o *elinks* como um exemplo) e que passos o programador deve dar para implementar essa funcionalidade em seus próprios aplicativos. Mas antes, teremos a costumeira aula de história…

## A lua como um todo

*Lua* começou em 1993 como uma linguagem para acadêmicos brasileiros da Tecgraf, na Pontifícia Universidade Católica do Rio de Janeiro (PUC-RJ). Sua proposta era fornecer um método simples de estender aplicativos através de uma linguagem procedural básica, tipos de dados tradicionais e uma máquina virtual. Todas essas características continuaram a ser parte fundamental de *Lua* até os dias de hoje, com a versão 5.0.2 (que contém correções para pequenos erros) no dia 17 de março de 2004. Sua assimilação foi uniforme (alguns diriam vagarosa), assim como seu ciclo de atualização; apenas 12 versões públicas em 11 anos de história. Em compensação, há muito pouca chance de quebra repentina do código, e todas as versões, até agora, têm sido incrivelmente estáveis.

A estrutura básica de Lua é a de um interpretador (o programa, *lua*) que gera e executa byte code, um conjunto de bibliotecas básicas opcionais (para entrada e saída, funções matemáticas e assim por diante, tudo escrito em C padrão) e um compilador (*luac*) para gerar byte code offline. Devido à natureza altamente padronizada da base de código de Lua, ela funciona em quase todas a plataformas que comportam um compilador C. Uma vez que é modular, pode encaixar-se em muitos dispositivos embutidos sem ocupar um espaço grande e desnecessário. A versão Linux, por exemplo, roda com apenas pouco mais de 100K; o conjunto de bibliotecas padrão incluído ocupa mais 72K. Muitos projetos embutidos tiraram vantagem de seu pequeno tamanho (e sua habilidade de compilar em qualquer lugar), como foi o caso de muitas empresas de jogos, como a Criterion Studios, a LucasArts (Grim Fandango), e a BioWare (com MDK2 e Baldur's Gate). Para uma lista mais longa dos usos, sugiro uma visita à referência [1].

## A lua cheia da primavera

Como linguagem, ela tem todas as características esperadas em uma linguagem funcional de script, incluindo o obrigatório manual online, disponível em [2]. Em primeiro lugar, nós temos os tipos de dados. Talvez não sejam abundantes (veja Tabela 1: Tipos em Lua), mas são

## Tabela 1: Tipos em Lua

| Tipo | Identificador | Nome para lua_*istype* |
|---|---|---|
| Nil | LUA_TNIL | nil |
| Number | LUA_TNUMBER | number |
| Bool | LUA_TBOOLEAN | boolean |
| String | LUA_TSTRING | string |
| Table | LUA_TTABLE | table |
| Function | LUA_TFUNCTION | cfunction |
| Userdata | LUA_TUSERDATA | userdata |

**Notas**

- O identificador pode ser usado para traduzir a constante em uma cadeia de caracteres dentro do C, usando *lua_typename(lua_State *L, int type)*
- *lua_isnumber* aceita os números (123) e cadeias de caracteres numéricos ("123")
- *lua_toboolean* retorna o para *false* e *nil*, e 1 para os demais

capazes de satisfazer às necessidades normais dos programadores. As variáveis em si seguem os moldes de uma linguagem flexível e podem comportar qualquer tipo, em qualquer momento. A tentativa de usar variáveis indefinidas faz com que elas contenham o tipo *nil*, que pode fazer com que certas operações (por exemplo, concatenação de cadeias de caracteres) falhem. O conceito de um "número" em Lua é equivalente ao tipo *double* da linguagem C. Todavia, viciados em desempenho que estejam usando a versão 5 podem modificá-lo para *float* (ou mesmo *int*) e recompilar *Lua*. Adicione:

```
#define LUA_NUMBER ➥
float
```

antes de incluir *lua.h*. Quem ainda utiliza a versão 4 terá que modificar mais um pouco o código.

Há também uma série tradicional de instruções de controle de fluxo, nos seguintes padrões:

```
do block end
while exp do block end
repeat block until exp
if exp then block end
if exp then block1 else block2 end
if exp then block1 elseif exp then ➥
block2 end
for var=start,end[,step] do block end
for var in iterator do block end
```

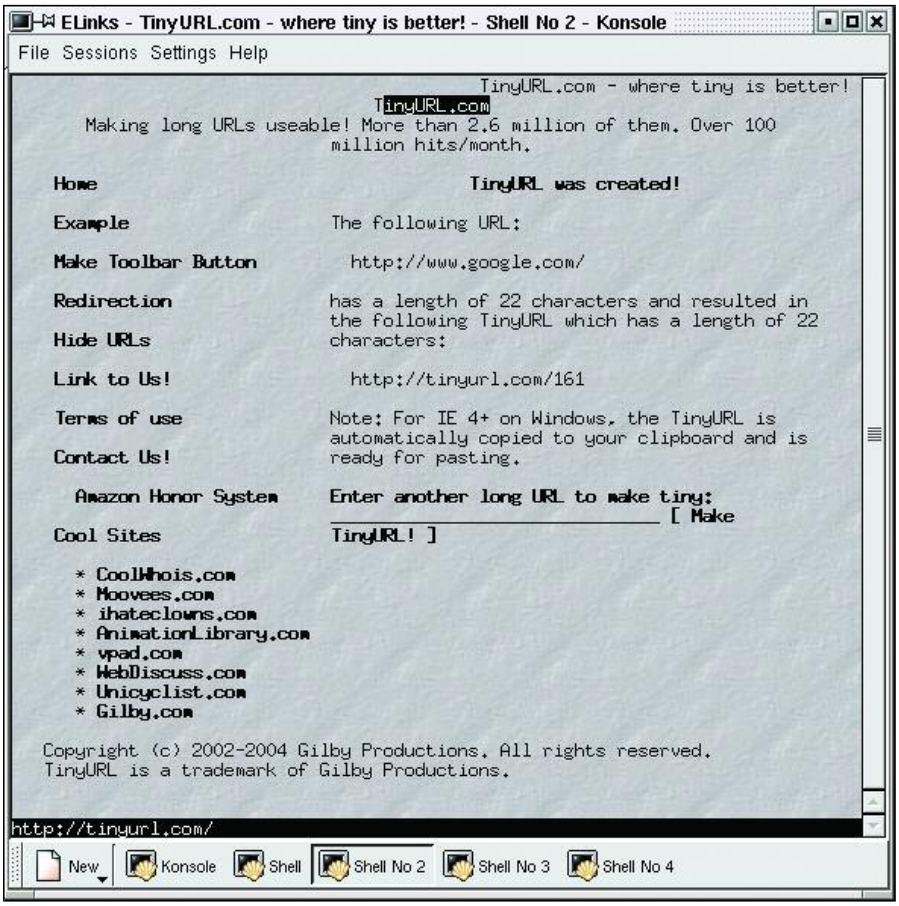


**Figura 1: O resultado da sua função gancho.**

Nada diferente aqui, mas programadores de C devem estar cientes de que o valor de 'end' é também calculado no loop *for*. Note que todas as instruções utilizam a palavra *end* como um finalizador de bloco, ao invés das chaves, mais tradicionais. Essa simplicidade é uma tentativa bastante óbvia de empurrar os não programadores para o mundo do script. A preferência por palavras ao invés de símbolos é também aparente quando você percebe que os operadores !, && e || foram substituídos pelas seus equivalentes alfabéticos, *not*, *and* e *or*.

A sintaxe também remove algumas das restrições encontradas em outras linguagens. Por exemplo, dois (ou mais) parâmetros podem ser retornados de funções, sem quaisquer problemas ou magia negra.

```
function onetwo()
return 1,2
end
one,two = onetwo()
```

Finalmente, em nosso rápido tour, Lua contém funções locais e anônimas. Uma função local é aquela que apenas pode ser chamada de dentro da função em que ela declaradamente está. Isso não é comum para os programadores de C, mas é bastante familiar para todos os outros. Já as funções anônimas permitem embutir uma função inteira no lugar em que nós normalmente colocaríamos uma chamada de retorno. Isso evita a necessidade de funções adicionais e arbitrárias dentro do código.

Para informações mais detalhadas sobre a sintaxe da linguagem, é um boa idéia ler a versão original disponível no website de Lua em [2]. O fórum de discussão em [3] está disponível para tópicos mais avançados. Porém, os fóruns podem receber um pouco de spam, de tempos em tempos.

## Listagem 1: hello.lua

```
01 // Insira aqui um comentário
   irônico sobre a originalidade
   do programador!
02 function hello()
03 write("Hello, Lua!")
04 end
05 hello()
```

Todavia, não são as características de Lua como linguagem o que a faz ser aprovada pelos desenvolvedores. É principalmente seu uso como ferramenta de configuração, que é considerado sua maior aplicação. É tão fácil para um desenvolvedor de aplicativos adicionar o suporte a scripts em Lua que seria possível perguntar por que a linguagem não é mais comum. Pode ser utilizada para criar plug-ins para softwares ou como um interpretador de arquivos de configuração. Não precisa ser uma configuração estática, como a maioria das outras aplicações. Ao utilizar uma linguagem de programação para gerar a configuração, você pode criar algo muito mais poderoso e flexível. Pode criar algo dinâmico!

Configurações dinâmicas têm sido um tanto raras. Apenas os aplicativos mais complexos, como o Apache, as possuem. Mesmo assim, diretivas como *IfModule* são reduzidas ao mínimo e têm um alcance limitado. Uma configuração verdadeiramente dinâmica pode facilitar o processo de instalação, preparando-se quando o programa é executado e adaptando-se de acordo com a estrutura do diretório, número de usuários, atual largura da banda, carga do processador e assim por diante.

*Lua* também disponibiliza um método fácil de adicionar ganchos (“*hooks*”) no software para personalização. Veremos esse atributo a seguir, adicionando alguns ganchos simples ao navegador de texto *elinks*.

## Caminhando à luz da lua

O gancho é um método através do qual um programa (nesse caso, o *elinks*) chama uma função especial toda vez que está para fazer alguma coisa importante. Esse ‘coisa importante’ pode ser ir para uma URL ou baixar uma página HTML do servidor. Em situações normais, sem gancho, essa função especial não fará nada! Nada mesmo! Ponto! Ela simplesmente devolverá o controle para o programa principal e o deixará ir para a URL para a qual estava indo anteriormente.

Entretanto, quando um gancho é colocado nessa função especial, o controle é passado do programa principal para a função gancho. Nessa altura, esta função tem o controle das informações e pode reescrever a URL, por exemplo. Como esses ganchos foram programados a partir de um programa Lua - o *nosso* - podemos reescrevê-los de acordo com nossas preferências pessoais.

**Tabela 2: Ganchos**

| Função Gancho | Chamada quando... | Deve retornar... | Notas |
|---|---|---|---|
| goto_url_hook(url, current_url) | Uma URL é digitada no diálogo “Go to URL” | Uma nova URL, ou zero para cancelar | |
| follow_url_hook(url) | Uma URL foi selecionada | Uma nova URL, ou zero para cancelar | |
| pre_format_html_hook(url, html) | Um documento foi baixado | Seqüência modificada, ou zero se nenhuma mudança foi feita | Pode Remover anúncios/lixo das webpages mal concebidas |
| lua_console_hook(string) | Algo é digitado no console de Lua (type , in *elinks*) | Um comando (run, eval, goto-url, or nil) seguido de um argumento apropriado (o programa para executar, o código Lua para validar, uma URL para ir, ou nulo, respectivamente) | |
| quit_hook() | *elinks* está para terminar | Nada | Para organizar os recursos |

Por exemplo, eu posso querer visitar um de meus próprios websites, *www.BlueDust.com*, digitando *bd*. Eu posso fazê-lo criando um gancho simples na rotina ‘Jump to URL’ com:

```
if (url == “bd”) then
return “www.bluedust.com”
end
```

Vejam só, configuração instantânea!

O programa usado para demonstração neste artigo, o *elinks*, tem uma variedade de ganchos diferentes, que podem ser vistos na Tabela 2: Ganchos e Frestas. Nós podemos utilizar um ou alguns deles para personalizar o aplicativo.

Não é só o *elinks* que chama nosso script, é possível para nosso script chamar o *elinks* através das chamadas de retorno. Isso nos permite resgatar informações, como a página de título, que não são repassadas como parâmetro. Isso é especialmente útil com teclas de atalho, como: quais parâmetros devem ser transferidos para elas? URLs? Listas de Favoritos? É aí que entram as funções de chamadas de retorno.

Essas chamadas de retorno são funções específicas que o *elinks* decidiu que nós (enquanto script) devemos utilizar. Ele nos permite chamá-las como se fizessem parte do próprio script em Lua. Incluem funções como *current_url* e *current_title*. Veja a lista na Tabela 3. Devemos agora empregar ambas as idéias para criar um gancho simples para o *elinks* que crie uma pequena URL para a página atual.

## A dança da lua

Nossa primeira tarefa é nos certificarmos de que o *elinks* foi de fato compilado com o suporte para scripts em Lua. Você pode conferir isso abrindo a caixa *About*: Pressionando *Alt + H*, seguido da letra *A*. Entre todo o texto, você deve encontrar as seguintes palavras:

```
Scripting (Lua)
```

**Quadro 1: enable_systems_functions**

openfile, closefile, readfrom, writeto, appendto, pipe_read, remove, rename, flush, seek, tmpname, read, write execute, exit, clock, date, getenv, setlocale.

**Listagem 2: *goto_url_hook***

```
01 function goto_url_hook (url, current_url)
02 if url == “tiny” then
03   return
   “http://tinyurl.com/create.php?url=”..current_url
04 end
05 return url
06 end
```

Isso já é normalmente incluído na maioria das distribuições (ou baixado em [4]). Casos o suporte não exista, corrija o problema recompilando o *elinks* com:

```
$./configure –with-lua
$ make
# make install
```

Isso também irá criar um arquivo de exemplo chamado *hooks.lua* no diretório *elinks/contrib/lua*. Depois, leia tudo com atenção e descubra algumas das outras possibilidades de script em Lua.

Em seguida, precisamos criar um script que será executado quando o *elinks* iniciar. Ele é colocado em *~/.elinks/hooks.lua* e é executado integralmente durante a inicialização. Então, desde que nosso código esteja dentro de *functions*, nada deve aparecer na tela.

A primeira de nossas funções irá incluir o código para *goto_url_hook*. Como foi anteriormente mencionado, ele é chamado toda vez que o usuário digita 'g' para mudar a webpage. É portanto bem simples escrever a Listagem 2.

É realmente fácil como aparenta! Recarregue o *elinks* e viste seu último website não favorito. Depois pressione 'g', seguido pela palavra-chave *tiny*, e retorne. Se você é como eu e escolhe o Google como seu site de teste, será levado para uma webpage em *tinyurl.com*, como mostrado na Figura 1.

Podemos ver a nova URL designada como *http://tinyurl.com/161*, que pode depois ser copiada, colada, enviada por e-mail e mal-utilizada de maneira geral. Se nós soubésssemos como adicionar atalhos no *elinks*, poderíamos poupar quatro teclas de atalho. Aqueles que leram anteriormente a Tabela 3, conhecerão a função chamada *bind_key*. Com ela, podemos adicionar o código mostrado na Listagem 3.

O exemplo demonstra a utilidade de funções anônimas e a facilidade com que dois valores podem ser retornados de uma função. No caso, o comando *goto_url* e o parâmetro URL.

Para melhorar o acabamento, vamos remover a informação duplicada da URL, escrevendo o arquivo *hooks.lua* como na Listagem 4.

Como você pode ver, Lua facilita ao usuário final adicionar funcionalidades em uma parte do software. Você não precisa de nada além dos métodos fornecidos aqui e um pouco de imaginação para adicionar um monte de outras funcionalidades. De um modo geral, dar mais flexibilidade ao usuário final significa complexidade para o programador. Com Lua, isso não é o que acontece! Vamos ver o porquê…

## Foguete Lunar

Em qualquer sistema extensível como esse, existem três componentes básicos. A inicialização (e desligamento), a comunicação de entrada (em que o script comunica-se com o nosso programa em C), e a comunicação de saída (em que o C comunica-se com o script). Todas as três áreas são muito simples e podem utilizar os modelos básicos mostrados aqui. Essa simplicidade manteve o interpretador Lua pequeno e encorajou os programadores a utilizarem-no em várias tarefas, como para configuração e automação de tarefas.

Vamos começar pelo início. Veja o exemplo a seguir:

**Listagem 3: Função *bind_key***

```
01 bind_key ("main", "Ctrl-T",
02   function ()
03     return "goto_url",
04 "http://tinyurl.com/create.php?url=".. current_url()
05   end
06   )
```

**Listagem 4: Arquivo *hooks.lua***

```
01 function get_tiny(url)
02   return "http://tinyurl.com/create.php?url=".. url
03 end
04 function goto_url_hook (url, current_url)
05 if url == "tiny" then
06   return get_tiny(current_url)
07 end
08 return url
09 end
10 bind_key ("main", "Ctrl-T", function () return ➥
   "goto_url", get_tiny(current_url() ) end )
```

```
#include <lua.h>
int main(int argc, char *argv[])
{
lua_State *pLua;
  pLua = lua_open();
  printf("Hello, World!");
  lua_close(pLua);
return 0;
}
```

A primeira coisa a ressaltar é que estou executando o Lua 5. Mesmo que essa seja uma preferência pessoal, pode causar pequenos problemas de compatibilidade, já que *lua_open* precisa de um parâmetro do tamanho da pilha na versão 4. Aqueles que executam a versão 4 (que inclui a versão estável do Debian) também irão notar que o arquivo de cabeçalho deve ser modificado para *lua40/lua.h*. Essas são duas das poucas mudanças incompatíveis com as versões

**Tabela 3: Funções**

| Nome da Função | Propósito |
|---|---|
| enable_systems_functions() | Permite que certas funções (por exemplo, abrir arquivo) sejam usadas. Veja Caixa 1. |
| current_url() | Recupera a URL da página atual do *elinks* |
| current_link() | Recupera o link que está selecionado no momento (ou *nil*, se nenhum) |
| current_title() | Recupera o título da página |
| current_document() | Recupera a página HTML, como uma cadeia de caracteres |
| current_document_formatted([width]) | Recupera a página HTML, formatada com o tamanho opcional |
| pipe_read(command) | Executa o *comando* dado e lê dados |
| execute(command) | Executa o *comando* dado em shell |
| | (using *sh -c*) |
| bind_key (keymap, keystroke, function) | Executa a *função* toda a vez que uma *tecla de atalho* é criada. Deve retornar um comando e um parâmetro como *l ua_console_hook* |

**Listagem 5: rotina *factorial***

```
01 bind_key (“main”, “Ctrl-T”,
02   function ()
03     return “goto_url”,“http://tinyurl.com/create.php?url=”..current_url()
05   end
06   )
```

anteriores feitas na versão 5. Ao compilar, devemos *linkar* nosso código C com a biblioteca de Lua da seguinte forma:

```
$ gcc -llua mycode.c
```

Se você está utilizando apenas Lua (isto é, uma versão standalone, não embutida em uma aplicação), precisará de suas próprias rotinas de entrada e saída. Isso não está incluído como padrão, uma vez que a maioria dos aplicativos provê suas próprias rotinas de E/S e, dessa forma, eliminam uma carga desnecessária. Para permitir ao código de Lua acessar suas bibliotecas padrão, inclua o seguinte código em C:

```
lua_baselibopen(pLua);
lua_iolibopen(pLua);
lua_strlibopen(pLua);
lua_mathlibopen(pLua);
```

A escolha das bibliotecas necessárias cabe, obviamente, a você. Todavia, incluir qualquer uma delas necessita vinculação com a biblioteca *lualib*. Utilizar a biblioteca de funções matemáticas de Lua requer a biblioteca de funções matemáticas do C (que não está incluída como padrão), e a linha de compilação vai se parecer com a seguinte:

```
$ gcc -llua -llualib -lm ➥
mycode.c
```

O *pLua* contém o estado de todo o sistema Lua. Uma vez que Lua é reentrante, podemos chamar *lua_open* quantas vezes nós quisermos, e nenhum estado irá entrar em conflito com o outro, o que nos permite usar Lua como parte de um sistema integrado. Toda vez que interagirmos com Lua, devemos utilizar esse ponteiro, que é convencionalmente chamado *L*, embora no texto eu esteja utilizando *pLua* para maior clareza.

Tendo acesso ao estado de Lua, podemos alimentá-lo com código para ser processado pelo interpretador interno:

```
lua_dostring(pLua, “number=12345”);
// other code
lua_dostring(pLua, “print(number)”);
```

Enquanto mantivermos o controle do *pLua*, quaisquer variáveis definidas permanecerão no estado de Lua. Toda vez que chamarmos uma função (como *print*), Lua irá validá-la com as funções declaradas e retornar os resultados para o estado mantido em *pLua*.

Provavelmente você pode imaginar como seria fácil construir seu próprio interpretador e depurador com apenas essa simples função, e você está certo! Porém, tal função já foi escrita para nós. É chamada *dofile*. Ela executa o código dentro do arquivo como se ele estivesse sendo executado a partir da linha de comando: isto é, executando apenas aquelas instruções que são globais. Todavia, diferentemente da linha de comando, assim que o arquivo for executado, o estado permanece em *pLua*. Essas variáveis, junto com quaisquer funções declaradas, podem ser acessadas pelo código em C, por outro arquivo Lua carregado com o *lua_dofile* ou pelo código validado com *lua_dostring*.

```
int result;
result = lua_dofile➥
(pLua, “config.lua”);
```

Nesse caso, *result* retorna o efeito do último segmento, que normalmente depende da execução bem-sucedida ou não o script.

## Luar sobre Parador

Vamos agora escrever algo útil em nosso script *config.lua*, que chama uma função em nosso programa em C. Isso teria o mesmo efeito que a função *current_url* no *elinks*, por exemplo.

Para fazer isso, precisamos registrar uma de nossas funções em C com Lua. Isso une as duas linguagens. Damos o nome que Lua usaria, seguido pelo nome atual de nossa função em C.

## Listagem 6: Chamando Lua

```
01 // O nome da função é um símbolo global: nós devemos usá-lo em vez ➥
   de uma cadeia de caracteres contendo o nome da função
02 lua_getglobal(pLua, “swap_greeting”);
03 // Nosso primeiro parâmetro
04 lua_pushstring(pLua, “Steev”);
05 // Nosso segundo parâmetro
06 lua_pushstring(pLua, “Hello”);
07 // A chamada em si
08 lua_call(pLua, 2/*number of input args*/, 2/*number of result args*/);
09 // Recuperar resultados dentro das variáveis de Lua
10 lua_setglobal(pLua, “result2”);
11 lua_setglobal(pLua, “result1”);
```

```
lua_register(pLua, “factorial”, ➥
c_factorial);
```

A partir daqui, podemos deixar que Lua tome conta. Ela organizará todas as chamadas de funções, passagem de parâmetros e o retorno dos resultados. Tudo o que nós precisamos fazer é recuperar os parâmetros na ordem certa e retornar os resultados corretos.

Já que Lua comporta múltiplos parâmetros de retorno (e de qualquer tipo), nós não podemos utilizar qualquer protótipo típico do C para administrá-la. No lugar disso, todos os parâmetros são enviados (e extraídos) de uma pilha. Nós podemos então consultá-la para saber quantos itens estão nela e de que tipo são. As pilhas, como variáveis, podem conter qualquer tipo de dado suportado. Cabe a nós, como programadores em C, requisitar o tipo apropriado ao remover dados da pilha. Isso garante que cada função do C tem a mesma assinatura, ou protótipo, quando usada com Lua.

Se nossa rotina *factorial* pode pegar um número arbitrário de números inteiros e retornar o fatorial de cada um, podemos recuperar o número de argumentos e computar o resultado para cada um - desde que seja númerico. Nosso código pareceria um pouco com o da Listagem 5.

Observe que os índices da pilha são contados a partir do 1 (não 0) e que precisamos retornar o número de parâmetros que são enviados para a pilha de retorno. Fora isso, não há mágica alguma em escrever suas próprias funções. Os tipos podem variar, de forma que *lua_isnumber* deve tornar-se *lua_isbool* (veja Tabela 1: Tipos de Lua), mas os princípios são exatamente os mesmos.

## Nuvens cruzando a lua

Chamar uma função de Lua a partir do C não é mais difícil, se você souber o padrão. Funciona com o mesmo princípio de antes, em que você envia dados para a pilha. Nesse caso, devemos colocar o nome da função primeiro, seguido por cada um dos argumentos em ordem. Uma vez que os tipos variam entre o C e Lua, você precisa utilizar a função correta para enviar o tipo apropriado para a pilha. Veja Listagem 6.

Isso teria o mesmo efeito que Lua chamando suas próprias funções; então,

```
result1, result2 = ➥
swap_greeting(“Steev”, “Hello”);
```

Observe que a ordem dos parâmetros é reversa, para combinar com sua remoção da pilha na ordem *First In, Last Out*.

Se quiséssemos os resultados dessa função no código em C, iríamos ler os dados da pilha e depois teríamos de removê-los explicitamente. Desta forma:

```
pResult2 = lua_tostring➥
(pLua, -1);
pResult1 = lua_tostring➥
(pLua, -2);
lua_pop(L, 2);
```

Repare na ordem reversa aqui também, e no uso de índices de pilha negativos.

## Shepherd Moons

Em poucas páginas, conseguimos aprender os fundamentos de uma nova linguagem de programação, adicionar novos recursos a um navegador Web e encontrar uma maneira de incrementar nossos próprios projetos com scripts dinâmicos de configuração. Tudo isso graças ao poder e à flexibilidade de Lua. Estou certo de que você também já tem suas próprias idéias de projetos onde Lua pode ser útil. Vá em frente! ■

SOBRE O AUTOR

*Quando operários vão a um bar, eles falam sobre futebol. Presume-se que, quando jogadores de futebol vão a um bar eles falam a respeito dos operários! Quando Steven Goodwin vai a um bar ele não fala sobre futebol ou operários. Ele fala sobre computadores. Invariavelmente...*

## Lua na terra dos mortos

Um dos maiores *cases* de sucesso no uso da linguagem Lua é o jogo *Grim Fandango*, um adventure da Lucas Arts. Segundo Bret Mogilefsky, programador chefe, originalmente o jogo deveria usar o *engine* SCUMM, o mesmo de todos os outros adventures da Lucas até a época. Contudo, SCUMM não era flexível o suficiente para as ambições do jogo e um novo engine teria de ser escrito. Procurando alternativas à tarefa de escrever um interpretador para uma nova linguagem de script, Bret encontrou Lua, na época na versão 2.50, que provou ser flexível e poderosa o suficiente para a tarefa, e acabou sendo usada em scripts para controlar diálogos e interações entre personagens, cenários e vários eventos do jogo. A linguagem também facilitou o processo de testes, já que no caso de um bug, os programadores podiam simplesmente chamar o interpretador Lua e fazer alterações no script em tempo real, sem ter de gerar um novo executável. Veja a entrevista [5] de Bret ao site grimfandango.net, onde ele conta mais detalhes sobre o uso de Lua.

## INFORMAÇÕES

[1] Usos de Lua: *http://www.lua.org/uses.html*

[2] Documentação de Lua: *http://www.lua.org/manual/5.0/*

[3] Fóruns sobre Lua: *http://archive.neotonic.com/archive/lua-l*

[4] Programa elinks: *http://elinks.or.cz/download.html*

[5] Entrevista com Bret Mogilefsky: *http://www.grimfandango.net/?page=articles&pagenumber=2*