The Lua-C API

Lua as a Library

- Lua is implemented as a library
- Exports ~90 functions
  - plus ~10 types, ~60 constants, ~20 macros
  - functions to run a chunk of code, to call Lua functions, to register C functions to be called by Lua, to get and set global variables, to manipulate tables, etc.
- Stand-alone interpreter is a small client of this library

A Naive Lua Interpreter

```
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main (int argc, char **argv) {
  lua_State *L = luaL_newstate();
  luaL_openlibs(L);
  luaL_loadfile(L, argv[1]);
  lua_call(L, 0, 0);
  lua_close(L);
  return 0;
}
```

Lua Kernel

```
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main (int argc, char **argv) {
  lua_State *L = luaL_newstate();
  luaL_openlibs(L);
  luaL_loadfile(L, argv[1]);
  lua_call(L, 0, 0);
  lua_close(L);
  return 0;
}
```

Auxiliary Library

```c
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main (int argc, char **argv) {
  lua_State *L = luaL_newstate();
  luaL_openlibs(L);
  luaL_loadfile(L, argv[1]);
  lua_call(L, 0, 0);
  lua_close(L);
  return 0;
}
```

needs malloc

needs file streams

Lua Libraries

```
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main (int argc, char **argv) {
  lua_State *L = luaL_newstate();
  luaL_openlibs(L);
  luaL_loadfile(L, argv[1]);
  lua_call(L, 0, 0);
  lua_close(L);
  return 0;
}
```

The Lua State

- All state of the interpreter stored in a dynamic structure `lua_State`
- State explicitly created (and destroyed) by the application
- All functions receive a state as first argument
  - except function to create a new state

The Lua State

```
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main (int argc, char **argv) {
  lua_State *L = luaL_newstate();
  luaL_openlibs(L);
  luaL_loadfile(L, argv[1]);
  lua_call(L, 0, 0);
  lua_close(L);
  return 0;
}
```

Multiple Lua States

- A state is completely self-contained
- A program can have multiple, independent Lua states
- Allows a lightweight implementation of Lua processes
  - each process is a Lua state
  - multiple workers (C threads) run those processes
  - communication through message passing

Lua Values

- Most APIs use some kind of “Value” type in C to represent values in the language
  - `PyObject` (Python), `jobject` (JNI)
- Problem: garbage collection
  - easy to create dangling references and memory leaks

Lua Values

- The Lua API has no `LuaObject` type
- A Lua object lives only inside Lua
- Two structures keep objects used by C:
  - the registry
  - the stack
- The *registry* is a regular Lua table always accessible by the API

The Stack

- Keeps all Lua objects in use by a C function
- Each function has its own private stack

Data Exchange

- The stack is the only channel for exchanging data between C and Lua
- *Injection functions*
  - convert a C value into a Lua value
  - push the result into the stack
- *Projection functions*
  - convert a Lua value into a C value
  - get the Lua value from anywhere in the stack
  - negative indices index from the top

Calling a Lua Function from C

- Push function, push arguments, do the call, get result from the stack

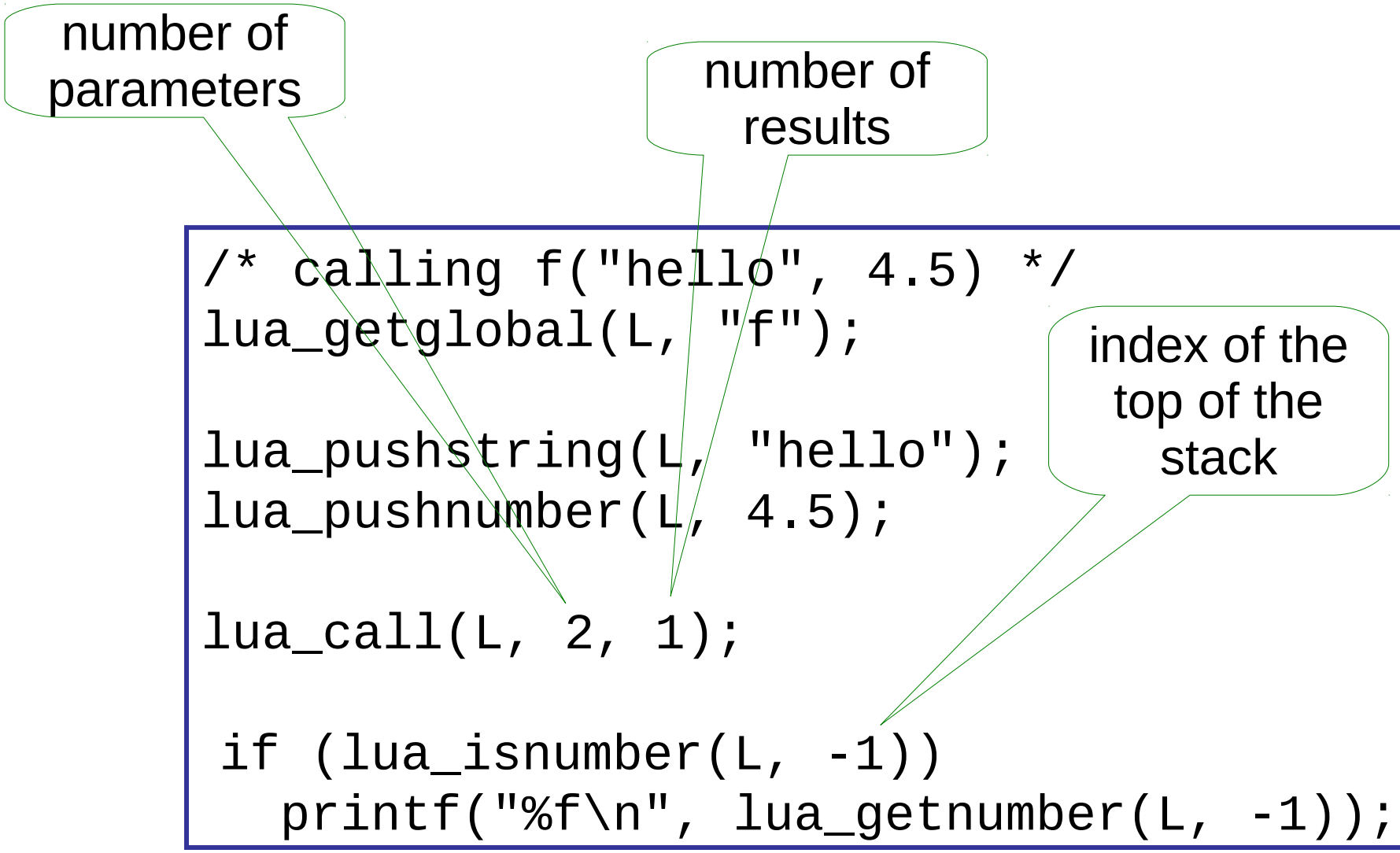
```
/* calling f("hello", 4.5) */
lua_getglobal(L, "f");

lua_pushstring(L, "hello");
lua_pushnumber(L, 4.5);

lua_call(L, 2, 1);

 if (lua_isnumber(L, -1))
   printf("%f\n", lua_getnumber(L, -1));
```

Calling a Lua Function from C



Calling a C function from Lua

- Function receives a Lua state (stack) and returns (in C) number of results (in Lua)
- Get arguments from the stack, do computation, push arguments into the stack

```
static int l_sqrt (lua_State *L) {
  double n = luaL_checknumber(L, 1);
   lua_pushnumber(L, sqrt(n));
   return 1;  /* number of results */
 }
```

Calling a C function from Lua

- Lua must know a C function to be able to call it
- Function `lua_pushcfunction` converts a C function into a Lua value in the stack
- After that, it is handled like any other Lua value

```
lua_pushcfunction(L, &l_sqrt);
lua_setglobal(L, "sin");
```

C “Closures”

- Lua can associate arbitrary Lua values to a C function
- When called by Lua, the C function can access these “upvalues”

Reflecting the API Back to Lua

- Lua offers most facilities through the API and exports them back to Lua scripts through libraries
  - `lua_pcall x pcall`
  - `lua_error x error`
  - `lua_resume x resume`
  - `...`