1 3 - The Application Program Interface

This section describes the C API for Lua, that is, the set of C functions available to the host program to communicate with Lua. All API functions and related types and constants are declared in the header file `lua.h` .
Even when we use the term "function", any facility in the API may be provided as a macro instead. All such macros use each of their arguments exactly once (except for the first argument, which is always a Lua state), and so do not generate any hidden side-effects.
As in most C libraries, the Lua API functions do not check their arguments for validity or consistency. However, you can change this behavior by compiling Lua with a proper definition for the macro `luai_apicheck` , in file `luaconf.h`.

1.1 3.1 - The Stack

Lua uses a *virtual stack* to pass values to and from C. Each element in this stack represents a Lua value (**nil**, number, string, etc.).
Whenever Lua calls C, the called function gets a new stack, which is independent of previous stacks and of stacks of C functions that are still active. This stack initially contains any arguments to the C function and it is where the C function pushes its results to be returned to the caller (see `lua_CFunction`).
For convenience, most query operations in the API do not follow a strict stack discipline. Instead, they can refer to any element in the stack by using an *index*: A positive index represents an *absolute* stack position (starting at 1); a negative index represents an *offset* relative to the top of the stack. More specifically, if the stack has *n* elements, then index 1 represents the first element (that is, the element that was pushed onto the stack first) and index *n* represents the last element; index -1 also represents the last element (that is, the element at the top) and index -*n* represents the first element. We say that an index is *valid* if it lies between 1 and the stack top (that is, if `1 &#8804; abs(index) &#8804; top`).

1.2 3.2 - Stack Size

When you interact with Lua API, you are responsible for ensuring consistency. In particular, *you are responsible for controlling stack overflow*. You can use the function `lua_checkstack` to grow the stack size.
Whenever Lua calls C, it ensures that at least `LUA_MINSTACK` stack positions are available. `LUA_MINSTACK` is defined as 20, so that usually you do not have to worry about stack space unless your code has loops pushing elements onto the stack.

Most query functions accept as indices any value inside the available stack space, that is, indices up to the maximum stack size you have set through lua_checkstack. Such indices are called *acceptable indices*. More formally, we define an *acceptable index* as follows:

```
(index < 0 \&\& abs(index) <= top) ||
     (index > 0 \&\& index <= stackspace)
```

Note that 0 is never an acceptable index.

1.3 3.3 - Pseudo-Indices

Unless otherwise noted, any function that accepts valid indices can also be called with *pseudo-indices*, which represent some Lua values that are accessible to C code but which are not in the stack. Pseudo-indices are used to access the thread environment, the function environment, the registry, and the upvalues of a C function (see §3.4).
The thread environment (where global variables live) is always at pseudo-index LUA_GLOBALSINDEX . The environment of the running C function is always at pseudo-index LUA_ENVIRONINDEX .
To access and change the value of global variables, you can use regular table operations over an environment table. For instance, to access the value of a global variable, do

```
lua_getfield(L, LUA_GLOBALSINDEX, varname);
```

1.4 3.4 - C Closures

When a C function is created, it is possible to associate some values with it, thus creating a *C closure*; these values are called *upvalues* and are accessible to the function whenever it is called (see lua_pushcclosure).
Whenever a C function is called, its upvalues are located at specific pseudo-indices. These pseudo-indices are produced by the macro lua_upvalueindex . The first value associated with a function is at position lua_upvalueindex(1), and so on. Any access to lua_upvalueindex(n), where *n* is greater than the number of upvalues of the current function (but not greater than 256), produces an acceptable (but invalid) index.

1.5 3.5 - Registry

Lua provides a *registry*, a pre-defined table that can be used by any C code to store whatever Lua value it needs to store. This table is always located at pseudo-index

`LUA_REGISTRYINDEX` . Any C library can store data into this table, but it should take care to choose keys different from those used by other libraries, to avoid collisions. Typically, you should use as key a string containing your library name or a light userdata with the address of a C object in your code.
The integer keys in the registry are used by the reference mechanism, implemented by the auxiliary library, and therefore should not be used for other purposes.

1.6 3.6 - Error Handling in C

Internally, Lua uses the C `longjmp` facility to handle errors. (You can also choose to use exceptions if you use C++; see file `luaconf.h`.) When Lua faces any error (such as memory allocation errors, type errors, syntax errors, and runtime errors) it *raises* an error; that is, it does a long jump. A *protected environment* uses `setjmp` to set a recover point; any error jumps to the most recent active recover point.
Most functions in the API can throw an error, for instance due to a memory allocation error. The documentation for each function indicates whether it can throw errors.
Inside a C function you can throw an error by calling `lua_error`.