Figure 1: uFAQ

Lua Unofficial FAQ (uFAQ)

Maintainer: Steve Donovan (steve j donovan at gmail com), 2009-2011; version 2.0.

1 Language

1.1 Where to start?

The official lua.org FAQ is here, which is the place to go for essential information like Lua availability and licensing issues.

There is also an incomplete Lua Wiki FAQ; this unofficial FAQ aims to fill in the holes and answer as many questions as possible, in a useful way.

Lua is a modern dynamic language with conventional syntax:

```
function sqr(x)
    return x*x
end

local t = {}
for i = 1,10 do
    t[i] = sqr(i)
    if i == 10 then
        print('finished '..i)
    end
end
```

Although at first glance it looks like Basic, Lua is more akin to JavaScript; such as no explicit class mechanism and equivalence of `a['x']` with `a.x`. There are very few types; string, number, table, function, thread and userdata. This simplicity accounts for a lot of Lua's speed and compactness compared with languages of equivalent power.

The Lua User's Wiki is full of useful example source and thoughtful discussion. In particular, here is a good place to start learning Lua.

The definitive book is Programming in Lua by Roberto Ierusalimschy, who is one of the creators of Lua. (the first edition is available online)

If you are coming to Lua from another programming language, then see the list of common gotchas.

And (of course), read the manual, although this is probably not a good place to *start*.

1.2 Suitability as a first programming language?

Lua is a compact language with a clean, conventional syntax which is easy to read. This makes it a good choice for embedding as a scripting language in larger applications, but also is suitable for introducing programming concepts.

Functions are first-class values and may be anonymous, so that functional styles can be learned. Proper closures and tail-call recursion are supported.

It runs interactively so exploring the language and its libraries is easy.

Some comparisons with other languages can be found on the Lua Wiki.

1.3 Suitability as a second programming language?

One scenario is that you already have big C/C++/Java/C# (etc) applications and wish to let your users script those applications. Lua will only add about 150-170K, which gets you a modern extension language which understands your special needs (like sand boxing) and can be easily integrated with your code.

The other is that you have a lot of components which you use in your work flow and you are looking for a glue language to easily bind them together quickly without messy recompilation. (See Ousterhout's Dichotomy.)

Another powerful use case which combines these two is embedding Lua as a *aid to debugging*. It is possible to provide an intelligent debug console (even for GUI applications, such as Java/Swing) running Lua, which can let you operate *inside* an application, querying state and running little test scripts.

1.4 Any good editors and debuggers?

See LuaEditorSupport.

Basically any of the common ones, including XCode.

SciTE is a popular editor for Windows and GTK+ platforms which has good Lua support. The version that ships with Lua for Windows can also debug Lua code. In addition, it is scriptable using its own embedded Lua interpreter.

There is a Lua plugin for IntelliJ IDEA.

1.5 Lua seems very verbose. Why isn't it like C?

The compactness of C (and the Unix environment in which it grew up) comes from the technical limitations of very clunky teleprinters. Now we have tab

completion and smart editors which can do abbreviations, so it doesn't really take more time to type Lua than C. For example, in SciTE one can add this to the abbreviations file:

```
if = if | then \n \
    \n \
end
```

Typing 'if' followed by `ctrl-B` will then put out the expanded text, properly indented.

Another point is that we spend much more time *reading* code than *writing* code, so the question is really whether it reads well. C is very readable to someone who is used to reading C, but Lua is more readable for casual programmers.

A C-like Lua derivative is Squirrel.

There is one important sense that Lua is the C of dynamic languages; it is small, fast, and only comes with the essential batteries. This is more than an analogy, since the Lua implementation only uses what is available with the C89 libraries.

1.5.1 Why do Lua arrays count from one? Oddly, the thing that really gets some people overexcited is that in Lua array indices count from one.

In mathematical notation array indices count from one, and so does Lua. In other words, they are not *offsets*.

The historical reason (see The Evolution of Lua) is that Lua evolved to solve engineering needs at the Brazillian state oil company (Petrobras). It was designed for engineers who were not professional programmers and more likely to be used to FORTRAN.

You *can* make your arrays start with zero, but the standard library functions (such as `table.concat`) will not recognise the zeroth entry. And `#` will return the size minus one.

```
t = {[0]=0,10,20,30}  -- table constructor is a little clunky
for i = 0,#t do print(i,t[i]) end  -- not 0,#t-1 !
```

1.5.2 Can I use a conditional expression like "x ? y : b" in C? The Lua form is `res = x and y or b` which works because the value of these operators is not just a boolean value, but is their second operand. Proper *short-circuiting* takes place, that is the expression `b` will not be evaluated if the condition was true.

But please note the case where `y` is either `nil` or `false`. The first `and` fails and we pick up `b` anyway.

Multiple values are understood:

```
> print (false and 5 or 10,'hello')
10      hello
```

People will often write a convenience function:

```
function choose(cond,a,b)
    if cond then return a else return b end
end
```

This evaluates both values first, so it is suitable only for simple values, but it *is* bulletproof.

1.6 How can undefined variable access be caught in Lua?

By default, an unknown variable is looked up in the global or module table and its value will be `nil`. This is a valid value for a variable, so mistyping a variable name can have interesting results, even if we were avoiding using globals like good boys and girls.

There are several approaches for enforcing some strictness with globals. The easiest is to check dynamically whether a given global variable has been initialised *explicitly* or not; that is, even if it has been set to `nil` somewhere (see `strict.lua` in the `etc` directory of the Lua distribution.)

The `tests/globals.lua` file in the distribution takes another approach, which is to pipe the bytecode listing of the compiler `luac` through `grep`, looking for any global access, but it's only useful if you 'avoid globals like the plague'. In particular, there are global functions and tables provided by the standard libraries.

lua-checker is a `lint`-like code checking tool.

LuaInspect provides real-time source code analysis for Lua using a [SciTE]((http://www.scintilla.org/SciTE.html) and a vim plugin; static output as HTML is supported. This provides *semantic highlighting* for Lua, so that for instance local variables are marked and can be visited, and unknown globals are flagged in red.

1.7 Does Lua support Unicode?

Yes and No. Lua strings can contain any characters (they are not NUL-terminated) so Unicode strings can be passed around without problems, as can any binary data. However, the existing string libraries assume single-byte characters, so a special library is needed; see slnunicode or ICU4Lua.

1.8 Optimization tips?

The first question is, do you actually have a problem? Is the program not *fast enough*? Remember the three basic requirements of a system: Correct, Robust and Efficient, and the engineering rule of thumb that you may have to pick only two.

Donald Knuth is often quoted about optimisation: "If you optimise everything, you will always be unhappy" and "we should forget about small efficiencies, say about 97% of the time: premature optimisation is the root of all evil."

Assume a program is correct and (hopefully) robust. There is a definite cost in optimising that program, both in programmer time and in code readability. If you don't know what the slow bits are, then you will waste time making your all your program ugly and maybe a little faster (which is why he says you will be unhappy.) So the first step is to use LuaProfiler the program and find the under-performing parts, which will be functions called very often and operations in inner loops. These are the only parts where optimisation is going to give you results, and since typically it is only a small part of the program, you end up doing a fraction of the work, with resulting happiness and less evil committed in terms of ugly code.

There are a few specific Lua tips here. If LuaJIT is available on your system, use that.

The equivalent in Lua of writing inner loops in Assembler is to factor out the heavy CPU-using code, write it as C, and use as an extension module. If done right, you will have a Lua program with almost the performance of a C program, but shorter and easier to maintain.

With LuaJIT you may not need to use C at all, especially since the built-in foreign-function interface FFI makes it very efficient to access external C functions and structures. This can also give more space-efficient representation for arrays of C types, compared to using tables.

1.9 When do I need to worry about memory?

The short answer is: only when you have to worry. Generally Lua does The Right Thing, but if memory use is excessive, then you do need to understand something about memory management in Lua.

People can get anxious about having a lot of string objects in memory. But Lua *interns* strings, so that there is really only one copy of each *distinct* string. So this code uses less memory than you would think:

```
local t = {}
for i = 1,10000 do
    t[i] = {firstname = names[i], address = addr[i]}
end
```

It also makes string equality tests very fast.

Excessive string concatenation can be slow and inefficient. Here is the wrong way to read a file into a string:

```
local t = ""
for line in io.lines() do
  t = t .. line .. '\n'
end
```

This is bad, for the same reason that doing this in Java or Python would be a bad idea; strings are immutable, so you are constantly creating and discarding strings.

If you *do* need to concatenate a lot strings, put them into a table and then stitch them together with `table.concat()`

Lua, like most modern languages, is garbage-collected. Memory allocated for tables, etc is automatically collected when the data is no longer *referenced*. Say we have `t = {1,2,3}` and later the assignment `t = {10,20,30}` happens; the original table is effectively orphaned, and gets collected next time the garbage collector comes around.

So the first thing to bear in mind is that data memory will only be freed if you genuinely have *no references* to that data. And the second is that it will generally not happen at the time of your choosing, but you can force a collection with `collectgarbage('collect')`. A subtle point here is that calling this function twice is often needed, because any finalizers will be scheduled for execution in the first pass, but will only execute in the second pass.

1.10 What is the difference between pairs and ipairs?

Lua tables are general data structures that can be viewed as both being 'array-like' and 'map-like'. The idea is that you can use them efficiently as resize-able arrays with integer indices, but can also index them with any other value as a hash map. `pairs(t)` gives a general iterator over all the keys and values in a table, numerical or not, in no particular order; `ipairs(t)` goes over the array part, in order.

Tables can contain a mixture of array and map entries:

```
t = {fred='one',[0]=1; 10,20,30,40}
```

The semi-colon is not required, it can be used as an alternative separator in table constructors. Just because `0` is a numerical index does not mean it is an *array index*! By definition, these go from `1` to `#t`.

These are equivalent ways to access the array elements:

```
for i,v in ipairs(t) do f(i,v) end

for i=1,#t do f(i,t[i]) end
```

Lua tables naturally express sparse arrays:

```
t = {[1] = 200, [2] = 300, [6] = 400, [13] = 500}
```

n this case, `#t` is unreliable, since then length operator `#` is only defined for non-sparse arrays (arrays without holes). If you wish to keep track of the size of a sparse array, then it is best to keep and increment a counter:

```
t[idx] = val;
t.n = t.n + 1
```

It is a little awkward to iterate over *only* the non-array part of a table:

```
local n = #t
for k,v in pairs(t) do
  if type(k) ~= 'number' or k < 1 or k > n then
    print(k,v)
  end
end
```

Why not have a special iterator for this? The reason is that arrays having array parts and hash parts is an *implementation* detail, not part of the specification. (It is straightforward to express the above as a custom iterator.)

1.11 Why have a special operator for string concatenation?

Having `..` as a separate operator reduces the opportunities for confusion.

Overloading '+' to mean string concatenation is a long tradition. But concatenation is not addition, and it is useful to keep the concepts separate, In Lua, strings can convert into numbers when appropriate (e.g 10+'20') and numbers can convert into strings (e.g `10 .. 'hello'`). Having separate operators means that there is no confusion, as famously happens in JavaScript.

When overloading operators, this distinction is also useful. For instance, a `List` class can be created where `..` again means concatenation, that is, joining different lists together to make a longer list. We could also define `+` for lists of numbers, and then it could mean element-wise addition, that is `v+u` is a list containing the sums of the elements of `v` and `u`, that is, they are treated as *vectors*.

1.12 I have seen Lua code with functions like strfind; why doesn't it work?

Functions like `strfind` mean that this is Lua 4.0 code; these functions are now in tables, like `string.find`. Look at the standard libraries section of the manual.

In Lua 5.1, all strings have a metatable pointing to the `string` table, so that `s:find('HELLO')` is equivalent to `string.find(s,'HELLO')`.

1.13 Why doesn't 'for k,v in t do' work anymore?

This is one of the few big changes between Lua 5.0 and 5.1. Although easy on the fingers, the old notation was not clear as to exactly how the key value pairs are generated. It was the equivalent of an explicit `pairs(t)`, and so isn't suitable for the common case of wanting to iterate over the elements of an array in order.

In the `for` statement, `t` must be a function. In the simplest case, it must be a function that returns one or more values each time it is called, and must return `nil` when the iteration is finished.

It is very straightforward to write iterators. This is a simple iterator over all elements of an array which works just like `ipairs`:

```
function iter(t)
  local i = 1
  return function()
     local val = t[i]
     if val == nil then return nil end
     local icurrent = i
     i = i + 1
     return icurrent,val
  end
end

for i,v in iter {10,20,30} do print(i,v) end
```

1.14 How can one put a nil value in an array?

Putting `nil` directly into an array causes a 'hole' that will confuse the length operator. There are two ways to get around this. One approach is a sparse array with an explicit size; the iterator must return the index and the value:

```
local sa = {n=0}

function append(t,val)
   t.n = t.n + 1
   t[t.n] = val
end

function sparse_iter(t)
  local i,n = 1,t.n  -- where t.n must be managed specially! #t will not work here.
  return function()
   local val = t[i]
   i = i + 1
   if i > n then return nil end
   return i,val
  end
end

append(sa,10)
append(sa,nil)
append(sa,20)

for i,v in sparse_iter(sa) do ... end
```

The other approach is to store a special unique value `Sentinel` in place of an actual `nil`; `Sentinel = {}` will do. These issues are discussed further on the wiki.

1.15 How can I dump out a table?

This is one of the things where a nice semi-official library would help. The first question is, do you want to do this for debugging, or wish to write out large tables for later reloading?

A simple solution to the first problem:

```
function dump(o)
    if type(o) == 'table' then
        local s = '{ '
        for k,v in pairs(o) do
                if type(k) ~= 'number' then k = '"'..k..'"' end
                s = s .. '['..k..'] = ' .. dump(v) .. ','
        end
        return s .. '} '
    else
        return tostring(o)
    end
end
```

However, this is not very efficient for big tables because of all the string concatenations involved, and will freak if you have *circular* references or 'cycles'.

PiL shows how to handle tables, with and without cycles.

More options are available here.

And if you want to understand a complicated set of related tables, sometimes a diagram is the best way to show relationships; this post gives a script which creates a Graphviz visualisation of table relationships. Here is what a typical result looks like.

1.16 Why does print 'hello' work, and not print 42?

The confusion arises because some languages (like Python 2.x) have a print *statement*, whereas `print` is a *function* in Lua (as it is n Python 3).

However, there are two cases where you can leave out the parentheses when calling a Lua function: you may pass a single value of either string or table type. So `myfunc{a=1,b=2}` is also fine. This little bit of 'syntactic sugar' comes from Lua's heritage as a data description language.

1.17 What is the difference between a.f(x) and a:f(x)?

`a.f` simply means 'look up f in a'. If the result is 'callable' then you can call it. A common pattern is to keep functions in tables, which is the approach taken by the standard libraries, e.g. `table.insert` etc.

`a:f` actually has no meaning on its own. `a:f(x)` is short for `a.f(a,x)` - that is, look up the function in the context of the object `a`, and call it passing the object `a` as the first parameter.

Given an object `a`, it is a common error to call a method using `..` The method is found, but the `self` parameter is not set, and the method crashes, complaining that the first parameter passed is not the object it was expecting.

1.18 How are variables scoped?

By default, variables are global, and are only local if they are function arguments or explicitly declared as `local`. (This is opposite to the Python rule, where you need a `global` keyword.)

```
G = 'hello'  -- global

function test (x,y)  -- x,y are local to test
  local a = x .. G
  local b = y .. G
  print(a,b)
end
```

Local variables are 'lexically scoped', and you may declare any variables as local within nested blocks without affecting the enclosing scope.

```
do
  local a = 1
  do
    local a = 2
    print(a)
  end
  print(a)
end

=>
2
1
```

There is one more scope possibility. If a variable is not local, in general it is contained inside the *module scope*, which usually is the global table (`_G` if you want it explicitly) but is redefined by the `module` function:

```
-- mod.lua (on the Lua module path)
module 'mod'

G = 1  -- inside this module's context

function fun(x) return x + G end --ditto

-- moduser.lua (anywhere)
require 'mod'
print(mod.G)
print(mod.fun(41))
```

1.18.1 Why aren't variables locally scoped by default? It certainly feels easy to only explicitly declare globals when they are in a local context. The short answer is that *Lua is not Python*, but there are actually good reasons why lexically-scoped local variables have to be explicitly declared. See the wiki page.

1.19 Difference between require and dofile?

`require` and `dofile` both load and execute Lua files. The first difference is that you pass the module name to `require` and an actual file path to a Lua file to `dofile`.

So what search path does `require` use? This is contained in the value of `package.path`: on a Unix system this would look like this (note that semicolons are used to separate items):

```
$> lua -e "print(package.path)"
./?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua;/usr/local/lib/
```

On a Windows machine:

```
C:\>lua -e "print(package.path)"
.\?.lua;C:\Program Files\Lua\5.1\lua\?.lua;C:\Program Files\Lua\5.1\lua\?\init.
lua;C:\Program Files\Lua\5.1\?.lua;C:\Program Files\Lua\5.1\?\init.lua;C:\Progra
m Files\Lua\5.1\lua\?.luac
```

It's important to note that Lua itself does *not* know anything about directories; it will take the module name and substitute it as '?' in each *pattern* and see if that file exists; if so, that is then loaded.

A very instructive error message happens if you try to require a module that does not exist. This shows you explicitly how Lua is trying to find the module by matching the path entries:

```
$> lua -lalice
lua: module 'alice' not found:
        no field package.preload['alice']
```

```
no file './alice.lua'
no file '/home/sdonovan/lua/lua/alice.lua'
no file '/home/sdonovan/lua/lua/alice/init.lua'
no file './alice.so'
no file '/home/sdonovan/lua/clibs/alice.so'
no file '/home/sdonovan/lua/clibs/loadall.so'
```

The second difference is the `require` keeps track of what modules have been loaded, so calling it again will not cause the module to be reloaded.

The third difference is that `require` will also load *binary modules*. In a similar fashion, `package.cpath` is used to find modules - they are shared libraries (or DLLs) which export an initialisation function.

```
$> lua -e "print(package.cpath)"
./?.so;/usr/local/lib/lua/5.1/?.so;/usr/local/lib/lua/5.1/loadall.so

C:\>lua -e "print(package.cpath)"
.\?.dll;.\?51.dll;C:\Program Files\Lua\5.1\?.dll;C:\Program Files\Lua\5.1\?51.dl
l;C:\Program Files\Lua\5.1\clibs\?.dll;C:\Program Files\Lua\5.1\clibs\?51.dll;C:
\Program Files\Lua\5.1\loadall.dll;C:\Program Files\Lua\5.1\clibs\loadall.dll
```

`package.path` and `package.cpath` can be modified, changing the behaviour of `require`. Lua will also use the environment variables `LUA_PATH` and `LUA_CPATH` if they are defined, to override the default paths.

What does './?.lua' match? This matches any Lua module in the *current directory*, which is useful for testing.

Why is there the pattern '/usr/local/share/lua/5.1/?/init.lua'? This is allows for self-contained packages of modules. So if there was a directory 'mylib' on the Lua path, then `require 'mylib'` would load 'mylib/init.lua'.

What if the module name contains dots, such as `require 'socket.core`? Lua replaces the period with the appropriate directory separator ('socket/core' or 'socket\core') and performs the search using this form. So on a Unix machine, it replaces '?' in '/usr/local/lib/lua/5.1/?.so' to give '/usr/local/lib/lua/5.1/socket/core.so' which matches.

1.20 How to explicitly load a binary module?

A binary module is a shared library that must have one entry point with a special name. For instance, if I have a binary module `fred` then its initialisation function must be called `luaopen_fred`. This code will explicitly load `fred` given a full path:

```
local fn,err = package.loadlib('/home/sdonovan/lua/clibs/fred.so','luaopen_fred')
if not fn then print(err)
else
  fn()
end
```

(For Windows, use an extension of `.dll`)

Note that *any* function exported by a shared library can be called in this way, as long as you know the name. When writing extensions in C++, it is important to export the entry point as `extern "C"`, otherwise the name will be mangled. On Windows, you have to ensure that at least this entry point is exported via `__declspec(dllexport)` or using a .def file.

1.21 What are function environments?

setfenv can set the environment for a function. Normally, the environment for a function is the global table, but this can be changed. This is very useful for sand boxing, because you can control the context in which code is executed, and prevent that code from doing anything nasty. It is also used to implement *module scope*.

```
function test ()
    return A + 2*B
end

t = { A = 10; B = 20 }

setfenv(test,t)

print(test())
=>
50
```

Function environments have been removed from Lua 5.2, but there are equivalent mechanisms of equal power available. Say you wish to compile some code in a particular starting environment; the extended load function can be passed code, a name for the code, the mode ("t" here for text only) and an optional environment table. `load` compiles the code, returning a chunk function that then needs to be evaluated:

```
Lua 5.2.0 (beta)  Copyright (C) 1994-2011 Lua.org, PUC-Rio
> chunk = load("return x+y","tmp","t",{x=1,y=2})
> = chunk
function: 003D93D0
> = chunk()
3
```

You can use this to load user scripts in a controlled environment; in this case the 'global' table just contains the specified two variables.

The Lua 5.1 `setfenv` example can be written using the special `_ENV` variable in Lua 5.2:

```
> function test(_ENV) return A + 2*B end
> t = {A=10; B=20}
> = test(t)
50
```

1.22 Can functions and operators be overloaded?

Functions cannot be overloaded, at least not at compile time.

However, you can 'overload' based on the types of the arguments you receive. Say we need a function which can operate on one or many numbers:

```
function overload(num)
  if type(num) == 'table' then
    for i,v in ipairs(num) do
      overload(v)
    end
  elseif type(num) == 'number' then
    dosomethingwith(num)
  end
end
```

The usual operators can be overloaded using metamethods. If an object (either a Lua table or C userdata) has a metatable then we can control the meaning of the arithmetic operators (like + - * / ^), concatenation .., calling () and comparison operators == ~= < >.

Overriding () allows for 'function objects' or *functors*, which can be used wherever Lua expects something that is callable.

Note a restriction on overloading operators like ==: both arguments must be of the same type. You cannot create your own `SuperNumber` class and get `sn == 0` to work, unfortunately. You would have to create an instance of `SuperNumber` called `Zero` and use that in your comparisons.

You can control the meaning of `a[i]` but this is not strictly speaking *overloading the [] operator*. `__index` fires when Lua *cannot* find a key inside a table. `__index` can be set to either a table or to a function; objects are often implemented by setting `__index` to be the metatable itself, and by putting the methods in the metatable. Since `a.fun` is equivalent to `a['fun']`, there is *no way* to distinguish between looking up using explicit indexing and using a dot. A naive `Set` class would put some methods in the metatable and store the elements of the set as keys in the object itself. But if `Set` a method 'set', then `s['set']` would always be true, even though 'set' would not be a member of the set.

The size operator and action when garbage-collected can be overridden by C extensions, not by plain Lua 5.1. This is restriction has been removed for Lua 5.2.

1.23 How to make functions receive a variable number of arguments?

Lua functions are very tolerant of extra arguments; you could take a function of no arguments and pass it arguments, and they will be simply discarded.

The syntax of functions taking an indefinite number of arguments ('variadic functions') is the same as in C:

```
function vararg1(arg1,...)
  local arg = {...}
  -- use arg as a table to access arguments
end
```

What is `...`? It is shorthand for all the unnamed arguments. So `function I(...) return ... end` is a general 'identity' function, which takes an arbitrary number of arguments and returns these as values. In this case, `{...}` will be filled with the values of all the extra arguments.

But, `vararg1` has a not-so-subtle problem. `nil` is a perfectly decent value to pass, but will cause a hole in the table `arg`, meaning that `#` will return the wrong value.

A more robust solution is:

```
function vararg2(arg1,...)
  local arg = {n=select('#',...),...}
  -- arg.n is the real size
  for i = 1,arg.n do
    print(arg[i])
  end
end
```

Here we use the `select` function to find out the actual number of arguments passed. This is such a common pattern with varargs that it has become a new function, `table.pack` in Lua 5.2.

You will sometimes see this in older code, which is equivalent to `vararg2`:

```
function vararg3(arg1,...)
  for i = 1,arg.n do
    print(arg[i])
  end
end
```

However, `arg` used in this special way is deprecated, and it will *not* work with LuaJIT and Lua 5.2.

1.24 How to return several values from a function?

A Lua function can return multiple values:

```
function sumdiff(x,y)
  return x+y, x-y
end

local a,b = sumdiff(20,10)
```

Note a difference between Lua and Python; Python returns several values by packing them into a single *tuple* value, whereas Lua genuinely returns *multiple values* and can do this very efficiently. The assignment is equivalent to the multiple Lua assignment

```
local a,b = 30,10
```

If your function constructs an array-like table, then it can return the array values as multiple values with `unpack`.

Another way for a function to return values is by modifying a table argument:

```
function table_mod(t)
  t.x = 2*t.x
  t.y = t.y - 1
end

t = {x=1,y=2}
table_mod(t)
assert(t.x==2 and t.y==1)
```

We can do this because tables are always passed by reference.

1.25 Can you pass named parameters to a function?

Named parameters are convenient for a function with a lot of parameters, particularly if only a few are commonly used. There's no syntax for named parameters in Lua, but it's easy to support them. You pass a table and exploit the fact that it is not necessary to use extra parentheses in the case of a function passed a single table argument:

```
function named(t)
  local name = t.name or 'anonymous'
  local os = t.os or '<unknown>'
  local email = t.email or t.name..'@'..t.os
  ...
end

named {name = 'bill', os = 'windows'}
```

Note the way that `or` works.

This style of passing named parameters involves creating a new table per call and so is not suitable for functions that will be called a large number of times.

1.26 Why is there no continue statement?

This is a common complaint. The Lua authors felt that `continue` was only one of a number of possible new control flow mechanisms (the fact that it cannot work with the scope rules of `repeat/until` was a secondary factor.)

In Lua 5.2, there is a goto statement which can be easily used to do the same job.

1.27 Is there any exception handling?

Lua programmers often prefer to return errors from functions. The usual convention is that if the function returns `nil` or `false` then the second value returned is the error message.

```
function sqr(n)
  if type(n) ~= 'number' then
    return nil,'parameter must be a number'
  else
    return n*n
  end
end
```

Of course, we know that in large applications it is often better to let the error immediately break execution and then to catch it outside. This allows us to write code that is not so cluttered with conditional error checking. Lua does allow a function to be called in a protected environment using `pcall`. Wrapping up the code in an anonymous function gives us the equivalent to a `try/catch` construct:

```
local status,err = pcall(function()
  t.alpha = 2.0   -- will throw an error if t is nil or not a table
end)
if not status then
  print(err)
end
```

This does seem a little clunky, and the temptation to find a cleaner syntax can be irresistible. MetaLua provides an elegant solution using compile-time macros.

One benefit of the explicit form here is that the cost of exception handling becomes clear; it involves creating a closure for the execution of each protected block.

The equivalent of `throw` or `raise` is just the function `error`.

The object 'thrown' by `error` can be any Lua object, but if it is not a string it should be convertible into a string. Also, Lua 5.1 does not apply `__tostring` to such error messages when reporting an error. Consider:

```
MT = {__tostring = function(t) return 'me' end}
t = {1,2}
setmetatable(t,MT)
error(t)
```

With Lua 5.1, the reported error is "(error object is not a string)" and with Lua 5.2 it is "me", as expected. This behaviour is not a problem with errors caught explicitly with `pcall`, since you can deal with the error object directly.

1.28 No classes? How do you guys survive?

Yes, there is no `class` statement in Lua, but doing OOP in Lua is not difficult. For a simple approach to single inheritance, see SimpleLuaClasses; for an overview, see ObjectOrientedProgramming. As a general piece of advice, pick a scheme, keep it simple, and be consistent.

Here is a sample from the first reference:

```
-- animal.lua

require 'class'

Animal = class(function(a,name)
   a.name = name
end)

function Animal:__tostring()
  return self.name..': '..self:speak()
end

Dog = class(Animal)

function Dog:speak()
  return 'bark'
end

Cat = class(Animal, function(c,name,breed)
         Animal.init(c,name)  -- must init base!
         c.breed = breed
      end)

function Cat:speak()
  return 'meow'
```

```
end

Lion = class(Cat)

function Lion:speak()
  return 'roar'
end

fido = Dog('Fido')
felix = Cat('Felix','Tabby')
leo = Lion('Leo','African')

D:\Downloads\func>lua -i animal.lua
> = fido,felix,leo
Fido: bark      Felix: meow     Leo: roar
> = leo:is_a(Animal)
true
> = leo:is_a(Dog)
false
> = leo:is_a(Cat)
true
```

Some method names are special, like `__tostring` ("metamethods"). Defining this function controls how our objects are represented as strings.

The `function TABLE:METHOD(args)` form is another little bit of syntactic sugar, it is *entirely* equivalent to `function TABLE.METHOD(self,args)`. That is, these are the same method definitions:

```
function Animal.feed(self,food)
...
end

function Animal:feed(food)
...
end
```

The beauty of Lua is that you have freedom to choose the *notation* you are comfortable with. For instance, we could define this syntax:

```
class 'Lion': inherit 'Cat' {
   speak = function(self)
     return 'roar'
   end
}
```

This seems like magic at first, but the first line is really `class('Lion'):inherit('Cat'):({`, so it cleverly uses the fact that Lua will accept single function arguments without parentheses if the argument is a string or a table. A 'class' object is

of course not a string, so the convention here is that classes are kept in the current module context (which would usually be the global table). One then has the opportunity to *name* the classes, which is a useful aid to debugging - for instance, it is then easy to give such classes a default `__tostring` which prints out their address *and* name.

Multiple Inheritance can be implemented, but there is a run-time cost compared to simple schemes.

On the other hand, here is an argument against needing inheritance in the first place: "In fully dynamic languages, where there's no compile-time type checking, there's no need to assure any pre-set commonality structure between similar objects. A given function can receive any kind of object, as long as it implements this and that methods".

The problem with not having a 'canonical' OOP scheme comes when integrating Lua code that uses an incompatible scheme. Then all you can assume is that an object can be called with `a:f()` notation. Re-use of classes via inheritance is only possible if you know how that class is structured. This can be considered a problem that hinders adoption of classic OOP style in Lua.

If inheritance is not an issue, the following pattern is a quick way to create 'fat objects':

```
function MyObject(n)
    local name -- this is our field
    local obj = {}  -- this is our object

    function obj:setName(n)
        name = n
    end

    function obj:getName()
        return name
    end

    return obj
end
...
> o = MyObject 'fido'
> = o:getName()
fido
> o:setName 'bonzo'
> = o:getName()
bonzo
```

The actual table returned only contains the two methods; the state of the object is *entirely* encapsulated in the closures. The *non-local variable* `name` (or 'upvalue') represents the object state.

Note that if this code was rewritten so that '.' was used instead of ':', then you can get *dot notation* for these objects, that is, `o.getName()`. This is however

not a good idea, since Lua programmers expect to use ':' and you may wish to change the implementation later.

Generally, this approach gives the fastest method dispatch at the cost of the method closures and their table per object. If there are only a few such objects, this cost may be worth paying.

1.28.1 What are closures?

A lot of the power of functions in Lua comes from *closures*. It helps to think of functions as dynamically-created objects, like tables. Consider this function which returns a function:

```
function count()
   local i = 0
   return function()
      i = i + 1
      return i
   end
end
```

Each time it is called, it returns a new closure.

```
> c1 = count()
> = c1()
1
> = c1()
2
> c2 = count()
> = c2()
1
> = c1()
3
```

For this to work, the variable `i` must be treated specially. If all the functions returned shared the same variable, then they would always return the next shared value of `i`. Instead, each function keeps its own copy of `i` - this variable is said to be an *upvalue* of the function. So one definition of closure is that it is a function plus any upvalues. This means something important: functions (like objects) can encapsulate state. This is one reason why the lack of a formal class concept is not seriously missed in Lua.

It allows for functional-style programming. Consider creating a new function with the first argument bound to some value (called *partial function application*)

```
function bind1(val,f)
    return function(...)
        return f(val,...)
    end
```

```
end
...
> print1 = bind1('hello',print)
> print1(10,20)
hello   10  20
```

Again, `f` and `val` are upvalues; every new call to `bind1` generates a new closure with its own references to `f` and `val`.

1.29 Is there string Interpolation, for example in "$VAR is expanded"?

Not directly supported, but easy to do with Lua's `string.gsub`.

```
res = ("$VAR is expanded"):gsub('%$(%w+)',SUBST)
```

`$` is 'magic', so it needs to be escaped, and the 'digits or letters' pattern `%w+` is captured and passed on to `SUBST` as 'VAR'. (A more correct pattern for a general identifier is '[_%a][_%w]*').

Here `SUBST` can either be a string, a function or a table of key-value pairs. So using `os.getenv` would allow us to expand environment variables and using something like `{VAR = 'dolly'}` would replace the symbol with the associated value in the table.

Cosmo packages and extends this pattern safely with sub-templates:

```
require "cosmo"
template = [==[
<h1>$list_name</h1>
<ul>
 $do_items[[<li>$item</li>]]
</ul>
]==]

print(cosmo.fill(template, {
    list_name = "My List",
    do_items  = function()
        for i=1,5 do
           cosmo.yield { item = i }
        end
    end
}
))
==>
<h1>My List</h1>
<ul>
<li>1</li><li>2</li><li>3</li><li>4</li><li>5</li>
</ul>
```

Python-like string formatting can be done in Lua:

```
print( "%5.2f" % math.pi )

print( "%-10.10s %04d" % { "test", 123 } )
```

The implementation is short and cool:

```
getmetatable("").__mod = function(a, b)
        if not b then
                return a
        elseif type(b) == "table" then
                return string.format(a, unpack(b))
        else
                return string.format(a, b)
        end
end
```

Lua strings share a metatable, so this code overrides the `%` (modulo) operator for all strings.

For this and other options see the String Interpolation page on the Wiki.

LuaShell is a very cool demonstration of how string interpolation and automatic function generation can give you a better shell scripting language:

```
require 'luashell'
luashell.setfenv()

-- echo is a built-in function
-- note that environment and local variables can be accessed via $
echo 'local variable foo is $foo'
echo 'PATH is $PATH'

cd '$HOME'    -- cd is also a built-in function

-- the ls function is dynamically created when called
-- the ls function will fork and exec /bin/ls
ls ()
ls '-la --sort=size'
```

1.30 What are optimised tail calls useful for?

Sometimes a problem can be naturally expressed recursively, but recursion seems too expensive, since usually all the stack frames have to be kept in memory.

This function defines an *infinite* loop in Lua:

```
function F() print 'yes'; return F() end
```

Consider this way of writing a Finite State Machine (FSM)

```
-- Define State A
function A()
  dosomething"in state A"
  if somecondition() then return B() end
  if done() then return end
  return A()
end

-- Define State B
function B()
  dosomething"in state B"
  if othercondition() then return A() end
  return B()
end

-- Run the FSM, starting in State A
A()
```

This machine is always in either state A or B, and can run *indefinitely* since the tail calls are optimised into something resembling a `goto`.

1.31 Packaging as a stand-alone application?

srlua does the basics; you can glue a Lua script to a Lua interpreter, and release that as a stand-alone executable.

However, most non-trivial scripts have dependencies; L-Bia aims to provide a more comprehensive solution.

A good solution is Squish which s a tool to pack many individual Lua scripts and their modules into a single Lua script, which can then be embedded using `srlua`.

1.32 How do I load and run (potentially untrusted) Lua code?

The task is to execute some code that comes from a (possibly) untrusted source.

```
-- make environment
local env = {}

-- run code under environment
local function run(code)
  local fun, message = loadstring(code)
  if not fun then return nil, message end
  setfenv(fun, env)
  return pcall(fun)
end
```

First the code is compiled into a function, then we set the environment for the function, and finally that function is called using `pcall`, so that we can catch errors. You could of course also use `loadfile` here.

Note that if you want to load code and *repeatedly* execute it, you should store the compiled function and `pcall` it whenever needed.

The code may come from anywhere. This is the classic 'sand-boxing' problem, (see Sandboxing.) The key to making such code as safe as possible is to restrict the environment of the function. Without the `setfenv` call, the function will run in the global environment and can of course call *any* Lua function, potentially causing a lot of damage. So the sand-boxing approach involves starting with an empty environment and only adding those functions which you want the user to use, excluding those which are 'unsafe'.

As for including your own modules in the sand boxed environment, note that `module('mymod',package.seeall)` is a definite security risk. This call works by creating a new environment in the table `mymod`, and then setting `__index` to be `_G` - so anybody can access global functions through your module, as simply as `mymod._G`! So resist the temptation to use `package.seeall`.

For Lua 5.2, an equivalent function would be:

```
function run (code)
   local fun, message = load(code,"tmp","t",env)
   if not fun then return nil, message end
   return pcall(fun)
end
```

1.33 Embedding Lua in an application?

One of Lua's most appreciated uses is in exposing select internal routines through Lua's efficient interpreter, such that users and developers may extend and customise an application's functionality. Lua is also an ideal candidate for modular cross-platform code within an application, or across its components. Applications that use Lua in these ways include Adobe Lightroom, World-of-Warcraft, VLC, Lighttpd, and more.

Embedding Lua will only add about 150-200K to your project, depending on what extra libraries are chosen. It was designed to be an extension language and it is straightforward to ensure that any user scripts operate in a 'safe' environment (see Sandboxing.) You do not even have to embed the compiler front-end of Lua, and just use the core with pre-compiled scripts. This can get the memory footprint down to about 40K.

The reasons that make Lua an ideal extension language are related to the common complaint from 'scripters' that Lua 'does not come with batteries'. The Lua core depends strictly on the facilities provided by ANSI C, so there are no platform-dependent bits; therefore, integrating it into your project is likely to be straightforward.

Integrating with C/C++ is especially easy (see 4.4 or Simple API Example), whilst bi-directional bindings for many popular languages and frameworks are also available (see Binding Code To Lua).

1.34 Documenting Lua code?

Code is as much an exercise in communicating with humans as it is with computers: as Donald Knuth says "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do". (After a few months, you are that human being that needs an explanation.)

LuaDoc is most commonly used. The style is similar to other documentation generation tools such as JavaDoc:

```
--- Three dashes mark the start; this sentence is the short summary.
-- You can add more lines here;
-- Note that they can contain HTML, and in fact you need an explicit <br>
-- to break lines.
-- @param first first name of person (string)
-- @param second second name of person (string)
-- @param age years; (optional number)
-- @return an object  (Person)
-- @usage person = register_person('john','doe')
-- @see Person
function register_person(first,second,age)
...
end
```

It is useful to comment your code in a structured way like this anyway, since Lua does not indicate the type of function arguments. If a user has to guess what to pass to a function by having to read the code then the function is not well documented. Having a *convention* for the type of the arguments is also useful.

However, auto-generated documentation is rarely enough. A set of well-documented tests does double duty in this respect.

1.35 How to check function argument types?

Dynamic typing leads to simple, general code. The function

```
function sqr(x) return x*x end
```

will work with any value that can be multiplied, i.e. either a number *or* an object which has defined the `__mul` metamethod. Also, since `type()` returns the actual Lua type, we can do various things depending on what we have been passed. Needless to say, this flexibility comes at a price, especially in a large code base. Good function documentation becomes essential.

Of course, the compiler does not read the documentation. There are type checking approaches which embed type information into Lua code. One style is to use `assert` on every function argument:

```
function sqr(x)
    assert(type(x) == "number", "sqr expects a number")
    return x*x
end
```

We now get an explicit meaningful error, at the cost of a lot of typing and a potentially significant runtime cost, In Lua, `assert()` is not a macro as it is in C, so it cannot be simply turned off. The typing can be reduced by defining suitable helper functions, but there are going to still be a lot of checks. Another objection is that the types of the function arguments are entirely contained in the function's implementation, and so cannot be accessed without some clever (and brittle) code analysis.

A promising approach is to use decorators:

```
sqr = typecheck("number","->","number") ..
function (x)
    return x*x
end
```

Now the *signature* of `sqr` is explicit and up-front, easily extract-able from the source, and actual runtime type checking can be turned off or on.

The type constraint 'number' is too specific, but one could define and use predicates like `is_number()`.

It is worth seriously considering Metalua, which is a smart, extensible Lua macro compiler (in the full Lisp sense of 'hygienic macro'). This syntax becomes possible:

```
-{ extension "types" }

function sum (x :: list(number)) :: number
    local acc :: number = 0
    for i=1, #x do acc=acc+x[i] end
    return acc
end
```

1.36 How to end script execution gracefully?

The function `error` will raise an error with the given message, and if it is not handled with `pcall` will result in a stack trace. This is useful when developing, but does not impress end users. Often people will define a function `quit` like so:

```
function quit(msg, was_error)
    io.stderr.write(msg,'\n')
    os.exit(was_error and 1 or 0)
end
```

`os.exit` terminates a script immediately, without invoking the garbage collector. (In Lua 5.2 there is an optional second argument to `os.exit` which *does* close the Lua state and force proper cleanup.)

The behaviour of `error` can be modified by setting `debug.traceback=nil` so that it doesn't produce a stack trace. (This is a nice example of Monkey patching; modifying Lua's operation by changing a library function dynamically.)

This monkey patch also affects the `assert` function. `assert` takes two arguments, a must-be-true condition and an error message. It is not a statement and returns the first argument. Because Lua functions often return a `nil` value and an error string when they fail, the following idiom works because the first argument passed to `assert` will be `nil`, and the second will be the error string.

```
f = assert(io.open(file))
```

Generally users of a module only want to see a stack trace involving their own code, and aren't interested in the internal details of the error (anyone who has used a badly-behaved Java library knows all about this.) The stack trace produced by `error` can be controlled by an optional 'level' parameter indicating who on the call stack is to be blamed for the error when generating the error message. A more general solution to this is provided by Lua Carp, modelled on the Perl `carp` module.

1.37 How does module() work?

module is used to create a module that can then be `require` d. Although not compulsory, it simplifies the creation of standard Lua modules.

```
-- testm.lua (somewhere on the Lua path)
local print = print
module ("testm")

function export1(s)
    print(s)
end

function export2(s)
    export1(s)
end
```

After `require ("testm")` you can then call the exported functions as `testm.export2("text")` etc. So the first thing `module` does is create a table

for your new functions; it also ensures that `require` will return this table (rather than just `true`)

Second, it sets the function environment of the chunk to be this table. This in effect makes it work like a 'namespace' in other languages; within the module `testm`, any function can see all other functions directly without qualifying with `testm.` This means that any global functions are not visible by default, which is why we had to create a local alias to `print`.

People can find this restrictive, so often modules are declared like so:

```
module ("testm",package.seeall)
```

This modifies the module table so that any unsatisfied look-up is done globally, so that the local declaration of `print` becomes unnecessary. (This is done by giving the table a metatable with `__index` pointing to the global table `_G`); note that accessing globals like this will be slower than explicitly declaring them as locals.

It is possible to write `module (...)` at the start of your module. When loaded, the module will be called with the name of the module as determined by `require`. This means that the module itself does not contain its name, and can be freely moved around.

The essential meaning of `module` is contained in this code snippet:

```
function mod(name)
   module ('test',package.seeall)
   function export()
    print 'export!'
   end
end

mod('test')

test.export()
```

That is, the `module` call makes the environment of the function `mod` to be `test`, so that any functions defined inside it are actually inside the `test` table.

1.37.1 Criticism of module() The first criticism is that `package.seeall` exposes the global table in cases where you really want to restrict access to functionality. It results in 'leaky encapsulation' which makes it easy for any user of this module to access global functions or tables through the module itself, e.g. `testm.io`.

The second criticism is that a module using `module` creates global tables and exports its dependencies. `module "hello.world"` creates a table `hello` (if not already present) and `world` as a table within that. If `hello.world` requires `fred` then `fred` becomes automatically available to all users of `hello.world`,

who may come to depend on this implementation detail and get confused if it changed.

See here.

Please note that `module()` has been deprecated for Lua 5.2; developers are encouraged to use a simple no-magic style for writing modules.

1.37.2 Life after module()? This module style remains portable between Lua 5.1 and Lua 5.2, where every exported function is explicitly put into a module table:

```
-- mod.lua
local M = {}

function M.answer()
  return 42
end

function M.show()
   print(M.answer())
end

return M
```

Note that no globals are created and in fact no module names are specified. So you need to assign the module to a variable before use:

```
local mod = require 'mod'
```

The disadvantage of the 'no-magic' style is that every module function reference has to be qualified, which can be a problem when converting code using `module()`. This alternative declares the functions up front:

```
local answer, show

function answer()  return 42 end

function show() print(answer()) end

return {answer=answer,show=show}
```

This also has the advantage that (a) exported functions are explicitly mentioned at the end and (b) all function accesses are local and hence faster. (Modules with a *lot* of functions will hit the local limit of about 240, but this is probably an excessive case.)

1.38 What are weak tables?

To understand weak tables, you need to understand a common problem with garbage collection. The collector must be conservative as possible, so cannot make any assumptions about data; it is not allowed to be psychic. As soon as objects are kept in a table, they are considered referenced and will not be collected as long as that table is referenced.

Objects referenced by a weak table will be collected if there is no other reference to them. If a table's metatable has a `__mode` field then it becomes weak; `__mode` may be one or both of 'k' (for weak keys) and 'v' (for weak values). Putting a value in a table with weak values is in effect telling the garbage collector that this is not a important reference and can be safely collected.

Note that strings should be considered a value (like numbers, unlike an object such as a table) for this purpose. This guarantees that string keys not be collected from a table with weak keys.

1.39 What is the difference between the various ways to quote strings?

There is no difference between single and double quoted strings, you may use either, although it's a good practice to be consistent. For example, people use single-quotes for internal strings and double-quotes for strings that are meant to be seen by the user/system.

These strings may contain C-style escape characters (like `\n`) although note that `\012` is the *decimal* constant 12! Long string quotes `[[..]]` do not interpret these escapes specially. As a convenience, the first line end is ignored in cases like this:

```
s = [[
A string with a single line end
]]
```

The price of this simplicity is that expressions such as `T[A[i]]` are not captured properly, so Lua allows for `[=[...]=]` where a matching number of `=` characters can be used. (This is the same scheme used with long comments, with a prefixed `--`.) Note that under Lua 5.0 nested `[[..]]` was allowed, but this has been deprecated.

Due to line-end processing these long string literals are not suitable for embedding binary data directly in programs.

1.40 Compatibility issues between Windows and Unix?

Here, 'Unix' stands for any POSIX-like operating system such as Linux, Mac OS X, Solaris, etc.

`package.config` is a string where the first 'character' is the directory separator; so `package.config:sub(1,1)` is either a slash or a backslash. As a general rule, try to use this when building paths.

A big difference between the Windows and Unix builds is that the default `package.path` is based on the location of the Windows executable, whereas on Unix it is based on `/usr/local/share/lua/5.1`. It is therefore easier to do a local user install of Lua on Windows, but Lua respects the environment variables `LUA_PATH` and `LUA_CPATH`.

Lua depends more directly on the system's C runtime libraries than most scripting languages so you have to appreciate platform differences. Use the "rb" specifier with `io.open` if you need compatibility with Windows binary I/O. Be careful of `os.tmpname` because it does not return a full path on Windows (prefix with the value of the `TMP` environment variable together with a backslash first.) `os.clock` is implemented very differently on Windows.

Likewise, `os.time` can actually crash Lua if passed non-compatible format specifiers. (This is no longer a problem with Lua 5.2, which does sanity checks first.)

For Windows GUI subsystem, `os.execute` can be irritating, and `io.popen` simply won't work - cross-platform extension libraries are available in this case.

2 Data

2.1 How do I access command-line arguments?

The command-line arguments are passed to a script in the global table `arg`. The actual arguments are `arg[i]` where `i` runs from `1` to `#arg` - or you could use `ipairs`. `arg[0]` is the name of the script, as it was called; `arg[-1]` is the name of the Lua executable used.

```
> lua arg.lua one "two three"
1       one
2       two three
-1      lua
0       arg.lua
```

(where 'arg.lua' is just 'for k,v in pairs(arg) do print(k,v) end')

You can use `arg[0]` to find out where your script is installed; if it's an absolute path, then use that, otherwise use the current directory plus the path given.

It is possible to make a directly executable script under Unix by making the first line '#!/usr/local/bin/lua' or '#!/usr/bin/env lua' and making the script itself executable. On Windows you can achieve this by adding ';.lua' to the PATHEXT environment variable and make execution the default action when opening Lua files.

Note that `arg` is set by the standard interpreter; if your C/C++ program launches a script with `luaL_dofile` then `arg` will be `nil`, unless you specifically set it.

2.2 How to Parse command-Line arguments?

There is no standard way to do this (which should come as no surprise), so packages tend to roll their own. However, the task is not hard using Lua string matching. Here is how LuaRocks parses its arguments:

```
--- Extract flags from an arguments list.
-- Given string arguments, extract flag arguments into a flags set.
-- For example, given "foo", "--tux=beep", "--bla", "bar", "--baz",
-- it would return the following:
-- {["bla"] = true, ["tux"] = "beep", ["baz"] = true}, "foo", "bar".
function parse_flags(...)
   local args = {...}
   local flags = {}
   for i = #args, 1, -1 do
      local flag = args[i]:match("^%-%-(.*)")
      if flag then
         local var,val = flag:match("([a-z_%-]*)=(.*)")
         if val then
            flags[var] = val
         else
            flags[flag] = true
         end
         table.remove(args, i)
      end
   end
   return flags, unpack(args)
end
```

Lapp provides a higher-level solution. It starts from the fact that you have to print out a usage string if you expect your script to be used by others. So, why not encode each allowable flag (with its type) in the usage string itself?

```
-- scale.lua
  require 'lapp'
  local args = lapp [[
  Does some calculations
    -o,--offset (default 0.0)  Offset to add to scaled number
    -s,--scale  (number)  Scaling factor
     <number> (number )  Number to be scaled
  ]]

  print(args.offset + args.scale * args.number)
```

Since the flags are *declared* as being numerical values, this very small script has fairly robust error checking:

```
D:\dev\lua\lapp>lua scale.lua
```

```
  scale.lua:missing required parameter: scale

  Does some calculations
   -o,--offset (default 0.0)  Offset to add to scaled number
   -s,--scale  (number)  Scaling factor
    <number> (number )  Number to be scaled

D:\dev\lua\lapp>lua scale.lua -s 2.2 10
  22

D:\dev\lua\lapp>lua scale.lua -s 2.2 x10
  scale.lua:unable to convert to number: x10
  ...
```

2.3 How to read all the lines in a file?

The simplest method is to use `io.lines`:

```
for line in io.lines 'myfile.txt' do
  ...
end
```

This form opens the file for you and ensures that it is closed afterwards. Without an argument, `io.lines()` gives you an iterator over all lines from standard input. Note that it is an iterator, this does not bring the whole file into memory initially.

It is better to open a file explicitly using `io.open`, since you can do proper error checking:

```
local f,err = io.open(file)
if not f then return print(err) end
for line in f:lines() do
  ...
end
f:close()
```

This is equivalent to:

```
local line = f:read()  -- by default, `read` reads a line
while line do
   ...
   line = f:read()
end
```

To read a whole file as a string, use the `*a` format with `read`:

```
s = f:read '*a'
```

2.4 How to read numbers?

The `read` method takes a list of format strings. So to read three numbers from each line, we have:

```
local f = io.open 'myfile.txt'  -- assuming it always exists!
while true do
  local x,y,z = f:read('*n','*n','*n')
  if not x then break end
  ...
end
f:close()
```

For more complicated situations, it is common to use `string.match` to extract the numbers and explicitly use `tonumber` to convert them.

2.5 How to format text and numbers nicely?

String concatenation works on both strings and numbers, but does not give you any control over the formatting of the numbers, which is particularly important when dealing with floating-point values.

The function `string.format` works like C's `sprintf`: you give it a string containing a format string, and a number of values. Numerical formats like '%5.2f' are understood (field width of 5, two places after the decimal point) or '%02d' (field width of 2, pad out with zeroes - '01', etc). '%s' means as a string, '%q' means as a *quoted string*, that is, a format that Lua can read back.

A useful little `printf` function:

```
function printf(s,...)
      print(string.format(s,...))
end
```

As always, your friend is the interactive prompt; experiment until you get the results you desire.

2.6 How to read CSV data?

Although in principle a simple format, the quoting rules can be awkward

The Penlight pl.data module can be useful here, but doesn't claim to be a full replacement for a specialised CSV module.

LuaCSV is a C module for reading CSV data. However, this Lpeg solution is just as fast, which is a surprising and interesting fact.

2.7 How to work with binary data?

Binary data can be handled as Lua strings, since strings are not NUL-terminated.

```
> s = 'one\0two\0three'
> = s
one
> = #s
13
> for k in s:gfind '%Z+' do print(k) end
one
two
three
```

Note that although the whole string could not be printed out, the length operator indicates that all bytes are present! `string.gfind` iterates over a string pattern; '%z' means the NUL byte, and '%Z' means 'anything except the NUL byte'.

Bear also in mind that '\ddd' escapes in Lua strings are *decimal*, not *octal* as they are in C. There is also no hexadecimal escape like '\xDD' although it is being considered for Lua 5.2

If I have a string containing the two bytes of a 16-bit integer, then `s:byte(2)+256*s:byte(1)` would do the job, *if* the byte order was little-endian.

To actually unpack non-trivial values from binary strings, a little help is needed. Roberto Ierusalimschy's struct library is useful here, since it knows about endianness and alignment issues:

For example, to pack a 32-bit float and a 16-bit bit integer into a string:

```
> s = struct.pack('hf',42,42)
> = #s
6
> = struct.unpack ('hf',s)
42      42      7
```

The last value returned is the point where we stopped reading, which in this case is beyond the end of the string. You can use this index to go through a array of structures.

2.8 Are there regular expression libraries?

Naturally! But see if you really cannot do the task with Lua's built-in *string patterns*. Although simpler than traditional regular expressions, they are powerful enough for most tasks. In most cases, you can understand a Lua string pattern by mentally replacing '%' with '\'; the advantage of % of course is that it does not itself need escaping in C-style strings such as Lua uses.

For example, extracting two space-separated integers from a string:

```
local n,m = s:match('(%d+)%s+(%d+)')
n = tonumber(n)
m = tonumber(m)
```

There is a string recipes page on the Wiki.

If you need more conventional regular expressions such as PCRE, see lrexlib.

Another library to consider instead of PCRE is Lpeg.

Here is a comparison of PCRE and Lpeg for a particular task:

2.9 Parsing complex structured text data?

If there is already a specialised parser for your particular data, use that - that is, don't create another XML parser!

For the special case of Windows .INI files and Unix configuration files, consider the Penlight pl.config module.

Lua is strong at text processing - you can use the built-in string patterns, PCRE regular expressions, or Lpeg.

Sometimes a lexical scanner gives a cleaner solution than pattern matching. This works particularly well if your 'complex structured text' is Lua or C code.

2.10 Are there mathematical libraries like complex numbers, higher-precision arithmetic, matrices, etc?

There are several ways to bring complex arithmetic into Lua. lcomplex allows you to write code like this:

```
> require 'complex'
> z = complex.new(3,4)
> = z*z
-7+24i
> = z + 10
13+4i
> = z:real()
3
> = z:imag()
4
> = z:conj()
3-4i
> = z:abs()
5
> = z^(1/3)  -- the cube root of z
1.6289371459222+0.52017450230455i
> i = complex.I
> = 10+4*i
10+4i
```

Another alternative, if complex arithmetic has to be as fast as possible and you don't mind working with a patched Lua, is the LNUM patch.

lbc provides arbitrary precision arithmetic based on the BC library:

```
> require 'bc'
> bc.digits()
> bc.digits(20)
> PI = bc.number "3.14159265358979323846"
> = PI
3.14159265358979323846
> = 2*PI
6.28318530717958647692
> = bc.sqrt(PI)
1.77245385090551602729
```

Matrix operations are provided by LuaMatrix

NumLua is a binding mostly based on the Netlib libraries.

GSL Shell is an interactive Lua binding to the GNU Scientific Library (GSL). It comes in two flavours; as a custom build of Lua using the LNUM patch for complex numbers and with a few enhancements, and as a extension library that can be called from vanilla Lua. It uses the powerful Anti-Grain Geometry (AGG) library to produce good looking graphs.

2.11 How to parse XML documents?

LuaExpat is the best known of the Lua XML bindings. This implements a SAX API where one registers callbacks to handle each element parsed.

The `lxp.lom` module brings an XML document into memory as a LOM table (Lua Object Model):

Given this XML fragment:

```
<abc a1="A1" a2="A2">inside tag `abc'</abc>
```

`lxp.lom.parse` will give you the following table structure:

```
{
    [1] = "inside tag `abc'",
    ["attr"] = {
        [1] = "a1",
        [2] = "a2",
        ["a1"] = "A1",
        ["a2"] = "A2",
    },
    ["tag"] = "abc",
}
```

The rules of LOM are simple:

- each element is a table
- The field `tag` is the element tag
- The field `attr` is a table of attributes; the array part has the attributes in order, and the map part has the mappings between attributes and values
- Any child nodes are in the array part of the element table.

2.12 How to interface with databases?

The GNU database manager (gdbm) keeps a set of key-value bindings, like a persistent hash table. lgdbm provides a simple API for managing gdbm databases:

```
d=gdbm.open("test.gdbm","n")
d:insert("JAN","January")
d:insert("FEB","February")
d:insert("MAR","March")
d:insert("APR","April")
d:insert("MAY","May")
d:insert("JUN","June")

for k,v in d:entries() do
  print(k,v)
end

d:close()
```

There is a `fetch` method for retrieving values, a `replace` method for updating entries, and a `delete` method for removing a key-value pair.

It is straightforward to put a more Lua-like facade on these basic operations:

```
local M = {
    __index=function (t,k) return t.gdbm:fetch(k) end,
    __newindex=function (t,k,v)
        if v then t.gdbm:replace(k,v) else t.gdbm:delete(k) end
    end
}

function gdbm.proxy(t)
 return setmetatable({gdbm=t},M)
end
```

Then things can be done in a more natural way:

```
t=d:proxy()
t.JUL="July"
```

```
t.AUG="August"
t.SEP="September"
t.OCT="October"
t.NOV="November"
t.DEC="December"

t.DEC="Dezembro"  -- replace the entry
t.SEP = nil  -- delete the entry
```

Usually people interpret the word 'database' as meaning 'relational database'.

Here is an example of using Lua-Sqlite3:

```
require "sqlite3"

db = sqlite3.open("example-db.sqlite3")

db:exec[[ CREATE TABLE test (id, content) ]]

stmt = db:prepare[[ INSERT INTO test VALUES (:key, :value) ]]

stmt:bind{  key = 1,  value = "Hello World"    }:exec()
stmt:bind{  key = 2,  value = "Hello Lua"      }:exec()
stmt:bind{  key = 3,  value = "Hello Sqlite3"  }:exec()

for row in db:rows("SELECT * FROM test") do
  print(row.id, row.content)
end
```

The LuaSQL project provides back-ends for the popular RDBMS engines. A little more involved, but providing much more database engine support:

```
require "luasql.postgres"
env = assert (luasql.postgres())
con = assert (env:connect("luasql-test")) -- connect to data source

res = con:execute"DROP TABLE people"
res = assert (con:execute[[
  CREATE TABLE people(
    name  varchar(50),
    email varchar(50)
  )
]])
-- add a few elements
list = {
  { name="Jose das Couves", email="jose@couves.com", },
  { name="Manoel Joaquim", email="manoel.joaquim@cafundo.com", },
  { name="Maria das Dores", email="maria@dores.com", },
}
```

```
for i, p in pairs (list) do
  res = assert (con:execute(string.format([[
    INSERT INTO people
    VALUES ('%s', '%s')]], p.name, p.email)
  ))
end

cur = assert (con:execute"SELECT name, email from people") -- retrieve a cursor

row = cur:fetch ({}, "a") -- want to index by field name
while row do
  print(string.format("Name: %s, E-mail: %s", row.name, row.email))
  row = cur:fetch (row, "a") -- reuse the table of results
end

cur:close()
con:close()
env:close()
```

2.13 Persisting program state?

First, see if writing to a standard format like CSV or XML best suits your needs; a database may also be more appropriate.

Second, Lua itself is a clean and fast format to encode hierarchical data. It is very straightforward to read Lua data and 'run' it to generate tables: use `loadfile()` to compile it, and run the resulting function.

Writing out Lua tables as Lua code takes a little more effort.

Finally, a Lua program can make its whole state persistent with Pluto.

3 Operating System

3.1 How to get the current date and time?

`os.date` without any arguments will give you a system-defined representation of the current date and time. You can pass it a formatting string just as the C function `strftime`:

```
> = os.date()
07/23/09 15:18:49
> = os.date '%A'
Thursday
> = os.date '%b'
Jul
> = os.date '%B'
July
```

```
> = os.date '%C'
> = os.date '%a %b %d %X %Y'
Thu Jul 23 15:21:38 2009
```

Note: the Microsoft implementation of `strftime` does not handle the full set of formatting specifiers. It is probably best, for cross-platform code, to stick to the codes specified here (some specifiers valid on Unix will actually cause Lua to crash on Windows - Lua 5.2 makes a sanity check here.):

The second, optional argument is the time to be formatted. This is the value returned by `os.time` and is (usually) the number of seconds since the start of the epoch, which is 1 Jan 1970.

`os.time` can be passed a table to be converted into a time value. See the manual for details.

3.2 How to execute an external command?

The simplest way is `os.execute` which works like the C `system` call; it calls out to the shell to execute shell commands and external programs. The return code of the execution is usually zero for success.

`io.popen` calls a command but returns a file object so you can read the output of the command, if the second argument is 'r', but you can also pass input to a command with a second argument of 'w'. Unfortunately, you don't get a `io.popen2`, and you don't get the return code.

There are also two serious weaknesses of these two functions when operating in Windows GUI mode; (a) you will get the dreaded black box flashing from `os.execute` and (b) `io.popen` just doesn't work.

A way around this is to use COM via LuaCOM. This is part of Lua for Windows

LuaCOM allows you to access the Windows Scripting Host (WSH) objects. The `Run` method is documented [here](http://msdn.microsoft.com/en-us/library/d5fk67ky(VS.85)

```
> require 'luacom'
> sh = luacom.CreateObject "WScript.Shell"
> = sh:Run ('cmd /C dir /B >  %TMP%\\tmp.txt',0)
```

The last argument hides the window that would otherwise be generated.

Another more lightweight option is using Winapi.

Be careful using `os.tmpname` on Windows; it will not return a full path to a temporary file; for that you need to prepend the value of the `%TMP%` environment variable. This used to be merely irritating, since a lot of temporary files ended up in the drive root, but under Vista this is frequently inaccessible (for good reasons).

3.3 How to make a script pause for a few seconds?

There is unfortunately no `os.sleep()`. But most platforms have a function call that can make a script suspend execution for a time period. This consumes no processor time, unlike a 'busy loop' which should be avoided.

LuaSocket provides a `socket.sleep` function which is passed the number of seconds to sleep; this is a floating-point number so it may be less than a second.

For Windows, Winapi provides a `sleep` function.

It is straightforward to bind to platform-specific sleep functions using LuaJIT FFI.

3.4 How to list the contents of a directory? Ask for file properties?

The bad news: Lua does not actually know about file systems. Lua is designed within the limits of what can be done in ANSI C, and that 'platform' doesn't know about directories. If this seems bizarre, bear in mind that Lua can and does run on embedded platforms that don't even have a file system.

You can use `io.popen` and directly use the shell to tell you what the contents of a directory is. But then you will have to keep cross-platform differences in mind (use 'ls' or 'dir /b'?) and so this gets messy. But it can be done - see LuaRocks as an example of a sophisticated Lua program which only leverages the standard libraries and the underlying shell for its operation.

A cleaner solution is to use an extension library like LuaFileSystem:

```
> require 'lfs'
> for f in lfs.dir '.' do print(f) end
.
..
lua
lrun
lual
lua51
scite
lscite
lrocks
luagdb
luajit
> a = lfs.attributes 'lua'
> for k,v in pairs(a) do print(k,v) end
dev     64772
change  1248283403
access  1248354582
rdev    0
nlink   1
blksize 4096
uid     1003
```

```
blocks  320
gid     100
ino     7148
mode    file
modification    1245230803
size    163092
```

Which tells me that 'lua' is a file, and has size of 163092 bytes. The time fields like 'modification' and 'access' can be translated into sensible dates using `os.date`.

If you prefer a higher-level library that harnesses LuaFileSystem, then look at pl.dir which provides a higher-level way to work with directories and provides convenience functions to copy or move files.

3.5 Running a program as a Service/Daemon?

On Windows, there is LuaService.

luadaemon will let your program become a daemon with a single `daemon.daemonize()` call; if it fails it will return an error string.

3.6 Has Lua got threads?

Yes and no, depending on the meaning of 'threads'. If you mean multitasking with OS threads, the answer is no, but Lua has *coroutines* which allow cooperative multitasking. A coroutine is a function which may *yield* at any point and may be *resumed* later. While it is yielded, it keeps its state, no matter how deep the call stack.

Since coroutines may be unfamiliar to most people, a little example:

```
local yield = coroutine.yield

function cofun()
    yield(10)
    yield(20)
    yield(30)
end

fun = coroutine.wrap(cofun)
print(fun())
print(fun())
print(fun())

=>
10
20
30
```

A powerful use of coroutines is to write *iterators*. It is often easy to express things such as tree traversal recursively, and using coroutines makes it almost trivial to wrap this as an iterator: `yield` can be called from any function called from a coroutine.

This does an 'in order' traversal of a tree data structure:

```
function tree(t)
  if t.left then tree(t.left) end
  yield (t.value)
  if t.right then tree(t.right) end
end

function tree_iter(t)
    return coroutine.wrap(function() return tree(t) end)
end

t = {
        value = 10,
        left = {
           value = 20
        },
        right = {
           value = 30
        }
}


for v in tree_iter(t) do
    print(v)
end

=>
20
10
30
```

The `for` statement is expecting a function of no arguments, which it will call repeatedly until `nil` is returned. `iter` wraps a function of no arguments which calls `tree` with the given tree table and returns it - an example of using a *closure*.

Copas uses coroutines in a particularly elegant way to allow the construction of network servers. The theory is nicely explained here in Programming In Lua.

```
-- Run the test file and the connect to the server using telnet on the used port.
-- The server should be able to echo any input, to stop the test just send the command "qui

require "copas"

local function echoHandler(skt)
```

```
  skt = copas.wrap(skt)
  while true do
    local data = skt:receive()
    if data == "quit" then
      break
    end
    skt:send(data)
  end
end

local server = socket.bind("localhost", 20000)

copas.addserver(server, echoHandler)

copas.loop()
```

Note that this example can handle a number of simultaneous connections, since Copas schedules the various sessions using coroutines. This is usually done using OS threads or ugly calls to `select()`.

There are libraries which take Lua beyond coroutines. True OS threads have been added to Lua with LuaThread, although this does require patching the source. Note that this can be inefficient due to the amount of locking required.

An efficient way to run multiple Lua states in parallel is provided by Lanes. These separate states are called 'lanes' and they do not share data, instead communicating with each with *linda* objects that act as channels.

A good overview of multitasking and concurrency in Lua is here, plus some comparisons

As for Lua and threading with C extensions, an introduction is given here and discussed further here. In particular, Lua coroutines can be safely driven from a different OS thread in a C extension.

3.7 How to access the Windows registry?

Have a look here.

Otherwise, use LuaCOM:

```
> require 'luacom'
> sh = luacom.CreateObject "WScript.Shell"
> = sh:RegRead "HKCU\\Console\\ColorTable01"
8388608
```

See the Microsoft Article for more details. This is also the easiest way to get network drives and connected printers, for instance.

Winapi has functions to read and write to the registry.

3.8 Sending an email?

This is easy with LuaSocket's SMTP module:

```
local smtp = require 'socket.smtp'

r, e = smtp.send {
    from = 'sjdonova@example.com',
    rcpt = 'sjdonova@example.com',
    server = 'smtp.example.com',
    source = smtp.message{
        headers = {
            subject = "my first message"
        },
        body = "my message text"
    }
}
if not r then print(e) end
```

Spam, unfortunately, has never been easier.

You can also use CDO with LuaCOM.

As for the other end, luaPOP3 provides a library for writing simple POP3 servers.

3.9 Can Lua be used as a Web scripting language?

Web applications built in Lua customarily perform better than their more common PHP and ROR brethren. Historically Lua has lacked good web development support with only basic CGI execution, however Lua now benefits from the broad functionality developed as the Kepler Project, including a MVC framework, SQL database drivers, templating, XML parsing, etc.

Orbit is a good framework for building web applications and Sputnik is an extensible Wiki framework for collaborative, document-based websites.

Serving Lua web applications can be done through FastCGI or vanilla CGI, with most web servers including Lighttpd, Apache, and Kepler's own Lua server, Xavante. WASAPI applications like Orbit or Sputnik can use any of these back-ends.

For something simpler to get started with you could instead try Haserl, a standalone CGI-based dynamic Lua page parser that you can just drop into the cgi-bin of any webhost (after you've asked them to install Lua).

3.10 Measuring elapsed time in a program?

If only second resolution is required, then `os.time()` is sufficient. For millisecond accuracy, there is `os.clock()`. However, there are two gotchas with this little function. The actual precision is likely to be much larger than a millisecond, and

the definition of 'process time' in Unix and Windows is fundamentally different. In Unix, `os.clock()` returns how much processor time has been used, so a script that spends most of its time sleeping will appear to use little time.

`socket.gettime()` returns fractional seconds, and is cross-platform.

4 Libraries

4.1 It's a compact language, but I need more libraries. Where to find them?

It is a deliberate decision on the part of Lua's designers to keep the core language compact and portable. It is understood that if people need libraries, they will write them. They feel that their job is to provide the 'kernel' and others will do the 'distribution', rather like how Linus Torvalds manages Linux. Hence the common remark that 'Lua does not include batteries'.

Since these libraries are not officially 'blessed' there does tend to be reinvention, but just about anything can be done with freely available Lua extension libraries.

A good place to start is the Libraries and Bindings page at the Lua Wiki.

LuaForge is the community repository of Lua libraries and hosts most of the good ones. A useful list of the most popular modules is available here.

You may prefer to download a distribution that will give you Lua plus a set of libraries. A good choice for Windows is Lua for Windows which includes SciTE for editing and debugging Lua, plus a lot of libraries plus documentation.

If you are running a Debian-based Linux like Ubuntu, then Lua and the common libraries are already available via apt-get or Synaptic. Going to rpmseek and entering 'lua' gives more than 200 hits, so generally you can find what you need using your OS' package system.

Another approach is LuaRocks which provides a cross-platform repository for downloading Lua packages. Like apt-get, it understands dependencies so that if you want LuaLogging it will also fetch LuaSocket; unlike apt-get, it works for Unix, OS X and Windows.

Another option is LuaDist which is pretty much an 'executable criticism' of LuaRocks. This may appeal to you if you're a fan of CMake.

4.2 What GUI toolkits are available?

This wiki page is a good start. In summary, all the major GUI libraries now have Lua bindings.

Probably the most full-featured and well-maintained binding is wxLua

There is a binding to Tcl/Tk. and also to Qt.

An easy and lightweight alternative for more casual scripting is IUP; both IUP and wxLua are part of Lua for Windows but they are available separately and are available for Unix as well.

Here is a simple plotting program:

```
require( "iuplua" )
require( "iupluacontrols" )
require( "iuplua_pplot51"  )

plot = iup.pplot{TITLE = "A simple XY Plot",
                    MARGINBOTTOM="35",
                    MARGINLEFT="35",
                    AXS_XLABEL="X",
                    AXS_YLABEL="Y"
                    }

iup.PPlotBegin(plot,0)
iup.PPlotAdd(plot,0,0)
iup.PPlotAdd(plot,5,5)
iup.PPlotAdd(plot,10,7)
iup.PPlotEnd(plot)

dlg = iup.dialog{plot; title="Plot Example",size="QUARTERxQUARTER"}

dlg:show()

iup.MainLoop()
```

With the iupx utility functions it can be as simple as this:

```
require "iupx"

plot = iupx.pplot {TITLE = "Simple Data", AXS_BOUNDS={0,0,100,100}}
plot:AddSeries ({{0,0},{10,10},{20,30},{30,45}})
plot:AddSeries ({{40,40},{50,55},{60,60},{70,65}})
iupx.show_dialog{plot; title="Easy Plotting",size="QUARTERxQUARTER"}
```

IUP is the most memory-efficient and fast GUI library reviewed here. For example, that last script takes just over 5 Meg of memory on Windows.

lua-gtk is a very comprehensive binding of Lua to GTK+ and other core libraries of the GNOME desktop.

```
require "gtk"

win = gtk.window_new(gtk.GTK_WINDOW_TOPLEVEL)
win:connect('destroy', function() gtk.main_quit() end)
win:set_title("Demo Program")
win:show()
gtk.main()
```

Another GTK binding is lgob.

luaplot is a plotting library for Lua, using Cairo and can be integrated with GTK+.

Depending on taste, using Lua with Java or .NET can be a productive way of generating GUIs. For instance, using LuaInterface you have direct access to the excellent NPlot plotting library. Using Lua with Java similarly makes it possible to create fully featured Swing applications.

4.3 Rendering graphics and plots to a file?

Lua-GD is a binding to the gd library. GD is particularly useful for Web applications, since it creates output in common formats like PNG, JPEG or GIF.

luaplot, mentioned above, can also render directly to graphic file formats.

luagraph is a binding to the Graphviz library, which allows you to visualise visualise complex graphs - that is, in the sense of a set of connected nodes.

This simple interface to gnuplot is useful if you are already familiar with this package and want to integrate it closer with your Lua code.

4.4 How to interface to my C/C++ libraries?

It is straightforward to bind Lua to C code. One reason is that Lua only has a few complex data types, so once you have learned to manage tables and userdata from the C side, you have mostly mastered the art of binding to Lua.

Arguments are taken off the stack, return results are pushed on the stack, and the C function returns the number of items it intends to return to the caller.

```
// build@ gcc -shared -I/home/sdonovan/lua/include -o mylib.so mylib.c
// includes for your code
#include <string.h>
#include <math.h>

// includes for Lua
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

// defining functions callable from Lua
static int l_createtable (lua_State *L) {
  int narr = luaL_optint(L,1,0);         // initial array slots, default 0
  int nrec = luaL_optint(L,2,0);   // intialof hash slots, default 0
  lua_createtable(L,narr,nrec);
  return 1;
}

static int l_solve (lua_State *L) {
    double a = lua_tonumber(L,1);  // coeff of x*x
    double b = lua_tonumber(L,2);  // coef of x
```

```
    double c = lua_tonumber(L,3);  // constant
    double abc = b*b - 4*a*c;
    if (abc < 0.0) {
        lua_pushnil(L);
        lua_pushstring(L,"imaginary roots!");
        return 2;
    } else {
        abc = sqrt(abc);
        a = 2*a;
        lua_pushnumber(L,(-b + abc)/a);
        lua_pushnumber(L,(+b - abc)/a);
        return 2;
    }
}

static const luaL_reg mylib[] = {
    {"createtable",l_createtable},
    {"solve",l_solve},
    {NULL,NULL}
};

int luaopen_mylib(lua_State *L)
{
    luaL_register (L, "mylib", mylib);
    return 1;
}
```

Note the convention; a library needs only to export one function, with the special name `luaopen_LIBNAME`. It will usually return the table created.

For this to work on Windows, you either must put `luaopen_mylib` into a .def file or use the modifier `__declspec(dllexport)` before the exported function definition. Because of Windows runtime issues, you will usually need to compile the extension with the same compiler that your version of Lua was itself built with.

This little module exports two functions, which are both useful (but have little otherwise to do with each other!). `mylib.solve` solves the quadratic equation and returns *both* roots; it properly detects the problem of imaginary roots and returns `nil` and an error message, which is the normal convention for errors.

`mylib.createtable` allows you to create a table with a *preset capacity* for new elements, which cannot otherwise be done from Lua itself. This can be useful if you have to fill a really big table, without taking the `O(log(N))` hit for inserting table elements.

```
> m = require 'mylib'
> = m.solve(0.5,10,1)
-0.10050506338833       -18.899494936612
> = m.solve(2,1,1)
nil     imaginary roots!
```

```
> t = m.createtable(20)
> for i=1,10 do t[i] = 2*i end
> = #t
10
```

Note that the table has initially *zero* size, and `#t` gets updated as new elements are inserted.

As a general rule, only go down to C if you have something that needs special optimization, or have an existing library that already does the job well.

Using the raw API is mostly straightforward, but can be tedious. There are several tools that will semi-automatically do the hard work for you, and they understand C++ as well. see 7.5.

4.5 Is there a foreign function Interface (FFI)?

Alien is a Lua binding to libffi.

Here is an example of binding to the C runtime; note that Windows is a special case requiring an explicit load. Alien handles the tedious part of binding to C for us:

```
require "alien"

if alien.platform == "windows" then
   libc = alien.load("msvcrt.dll")
else
   libc = alien.default
end

libc.malloc:types("pointer", "int")
libc.free:types("void", "pointer")
libc.strcpy:types("void", "pointer", "string")
libc.strcat:types("void", "pointer", "string")
libc.puts:types("void", "string")

local foo = libc.malloc(string.len("foo") + string.len("bar") + 1)
libc.strcpy(foo, "foo")
libc.strcat(foo, "bar")
libc.puts(foo)
libc.strcpy(foo, "bar")
libc.puts(foo)
libc.puts("Yeah!")
libc.free(foo)
```

Here is the classic GTK+ “Hello, World”:

```
local gtk,p,i=alien.load("/usr/lib/libgtk-x11-2.0.so.0"),"pointer","int"
```

```
gtk.gtk_init:types(nil,p,p)
gtk.gtk_message_dialog_new:types(p,p,i,i,i,p)
gtk.gtk_dialog_run:types(i,p)

gtk.gtk_init(nil,nil)
gtk.gtk_dialog_run(gtk.gtk_message_dialog_new(nil,0,0,1,"Alien Rocks!"))
```

People can get a little bit overexcited at this point and to try doing non-trivial GTK+ applications with Alien, but there are a *lot* of functions and constants involved; you are better off with a full-featured GTK binding like lua-gtk. But if you just want your script to put up a message, it works fine.

One issue that may bite you with Alien is that you need the full path to the shared library on Unix (Windows has a more relaxed shared library path) and these paths will move around depending on the whim of the distribution maintainers. For example, the shared libraries are in `/usr/lib64` on RedHat/SuSE x86_64 distributions (but not in Debian/Ubuntu).

How fast is Alien? Calling `sin` through Alien takes about 3 times longer than `math.sin`. So there is a definite cost for accessing functions in this way, but in many cases the actual work done by the function is much larger than the call overhead.

4.5.1 LuaJIT FFI LuaJIT's built-in FFI provides a very fast way for Lua programs to access C functions and structures. Provided LuaJIT supports your platform, it will be as fast (if not faster) than writing C extensions. Simple C function and structure declarations are directly understood:

```
ffi.cdef[[
void Sleep(int ms);
]]
```

after which the function can be called directly as `ffi.C.sleep`.

C arrays and structures can be directly created and accessed - note that arrays are *zero-based*:

```
> ffi = require 'ffi'
> arr = ffi.new("double[?]",20)
> for i = 1,20 do arr[i-1] = i end
> = arr[10];
11
 > ffi.cdef [[struct _point {double x,y;};]]
 > point = ffi.new("struct _point")
> point.x, point.y = 10,20
> = point.x, point.y
10      20
```

5 Hardware

5.1 Smart Phones

5.1.1 Google Android? There is direct support for Lua scripting with Android; see this announcement on the Google Open Source blog.

AndroLua enables using LuaJava to integrate regular C Lua into Android projects.

Corona allows for cross-platform Android and iPhone development in Lua.

5.1.2 IPhone Support? Apple no longer disallows interpreters on the iPhone and there is no reason why an application cannot contain Lua; the restrictions seem to be against downloading and running scripts that access the iPhone APIs. You can see an iPhone running 'Hello world!' here.

There is a relevant question answered on StackOverflow.

Mathew Burke discusses options for iOS programming in Lua in this luanova article, including Corona and Wax.

5.1.3 Java-based Phones? Kahlua is a project to completely implement the Lua engine in Java. The target platform is the reduced runtime J2ME found in mobile phones; the main example is a little Lua interpreter which can run on your phone. Another option is luaj.

Jill (Java Implementation of Lua Language) has been recently released.

5.2 What are the minimum requirements for an embedded device?

Lua is not a classic interpreter; source code is compiled to binary form and then executed, which the `luac` tool will do explicitly for you. Without the compiler, the Lua VM is then only about 40K. The binary code can be converted into C strings and compiled directly into your application, which is necessary if you do not actually *have* a file system.

eLua is a project dedicated to Lua running 'unhosted' on embedded processors. The author says "I recommend at least 256k of Flash .. and at least 64k of RAM', although this is for floating-point support. The fact that Lua can be complied to use a different numerical type is very useful here, because many microprocessors have no built-in floating support.

The LNUM patch by Asko Kauppi allows you to have the best of both worlds, by allowing both integer and floating-point types to coexist in Lua. This works very well on the ARM-based processors on Gumstix boards.

6 Interoperability

6.1 LuaJIT is significantly faster than vanilla Lua. What's the catch?

The recent release of LuaJIT 2 establishes LuaJIT as the fastest dynamic language implementation. 32/64 bit Intel and ARM processors are supported, and performance can be comparable to compiled code.

There are some differences you should be aware of. LuaJIT is stricter than vanilla Lua and does not support the old Lua 5.0 way of dealing with a variable number of arguments.

LuaJIT now supports the dumping and reading of bytecode, but currently libraries such as Lanes cannot work with it currently. This applies to other extensions like Pluto which need deep integration with the Lua implementation.

6.2 Using Lua with .NET, Java or Objective-C?

Typically, the need is for a compact scripting language that can be easily embedded in a larger application.

With the VM platforms, there are two approaches: one is to bind the Lua C library with JNI or P/Invoke, and the other is to run Lua entirely in 'managed code' If you are familiar with these platforms, Lua provides an easy way to explore the available libraries and prototype user interfaces. The quest for extra batteries then becomes the question 'Can I do it in C# or Java?'.

6.2.1 Lua with Java LuaJava allows Lua scripts to run within Java. The Lua interpreter (plus some binding code) is loaded using JNI.

Here is an example script:

```
sys = luajava.bindClass("java.lang.System")
print ( sys:currentTimeMillis() )
```

(Note that there is no `require 'luajava'` needed because we are running within LuaJava.)

`newInstance` creates a new instance of an object of a named class.

```
strTk = luajava.newInstance("java.util.StringTokenizer",
    "a,b,c,d", ",")
while strTk:hasMoreTokens() do
    print(strTk:nextToken())
end
```

Which can also be expressed as

```
StringTokenizer = luajava.bindClass("java.util.StringTokenizer")
strTk = luajava.new(StringTokenizer,"a,b,c,d", ",")
...
```

LuaJavaUtils is a little set of utilities to make using LuaJava easier. The `java.import` module provides an `import` function which works like the Java statement of this name.

```
require 'java.import'
import 'java.util.*'
strTk = StringTokenizer("a,b,c,d", ",")
...
```

This module modifies the global Lua environment so that undeclared globals are first looked up in the import list, and made into class instances if possible; the `call` metamethod is also overloaded for these instances so we can say `Class()` rather than `luajava.new(Class)`.

There are also a few little Java classes to get around a limitation of LuaJava, which is that one cannot inherit from full classes.

```
-- a simple Swing application.
require 'java.import'
import 'java.awt.*'
import 'java.awt.event.*'
import 'javax.swing.*'
import 'java.awt.image.*'
import 'javax.imageio.*'
import 'java.io.*'
-- this is for PaintPanel and Painter...
import 'org.penlight.luajava.utils.*'

bi = ImageIO:read(File(arg[1] or "bld.jpg"))
w = bi:getWidth(nil);
h = bi:getHeight(nil);

painter = proxy("Painter", {
    painter = function(g)
        g:drawImage(bi, 0, 0, nil)
    end
})

f = JFrame("Image")
f:setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)

canv = PaintPanel(painter)
canv:setPreferredSize(Dimension(w,h))

f:add(JScrollPane(canv))
f:pack()
f:setVisible(true)
```

LuaJava only supports Java 1.4, so nifty features like generics and variable argument lists are not supported. Also, currently error handling is a little primitive.

Kahlua is a project to completely implement the Lua engine in Java. The target platform is the reduced runtime J2ME found in mobile phones; the main example is a little Lua interpreter which can run on your phone.

Another option is luaj which targets both J2ME and J2SE and "unique direct lua-to-java-bytecode compiling".

6.2.2 Lua with .NET LuaInterface is a binding of Lua to .NET; up to version 1.5.3 using P/Invoke, since version 2.0 fully managed code. The 1.5.3 version is more friendly to the more casual user of .NET (say a developer wishing to use a WinForms interface for Lua) and so this is the version included in Lua for Windows.

A console "Hello, World!" using LuaInterface:

```
require 'luanet'
luanet.load_assembly "System"
Console = luanet.import_type "System.Console"
Math = luanet.import_type "System.Math"

Console.WriteLine("sqrt(2) is {0}",Math.Sqrt(2))
```

A basic Windows.Forms application:

```
require 'luanet'
luanet.load_assembly "System.Windows.Forms"
Form = luanet.import_type "System.Windows.Forms.Form"
form = Form()
form.Text = "Hello, World!"
form:ShowDialog()
```

Since all types need to be explicitly loaded, this can get a bit tedious. A helper library CLRPackage (code) written in pure Lua can reduce the amount of typing by providing an import facility:

```
require 'CLRPackage'
import "System"
Console.WriteLine("sqrt(2) is {0}",Math.Sqrt(2))
```

Windows.Forms applications become more straightforward:

```
require 'CLRPackage'
import "System.Windows.Forms"
import "System.Drawing"

form = Form()
form.Text = "Hello, World!"
button = Button()
```

```
button.Text = "Click Me!"
button.Location = Point(20,20)
button.Click:Add(function()
    MessageBox.Show("We wuz clicked!",arg[0],MessageBoxButtons.OK)
end)

form.Controls:Add(button)
form:ShowDialog()
```

GUI toolkits tend to split into two camps; those that use absolute positioning (like `Point(20,20)` above) and those which believe that this is Evil and use widget-packing schemes (like Java and GTK). Here is an alternative way to construct forms:

```
-- autoform.wlua
require "CLRForm"

tbl = {
    x = 2.3,
    y = 10.2,
    z = "two",
    t = -1.0,
    file = "c:\\lang\\lua\\ilua.lua",
    outfile = "",
    res = true,
}

form = AutoVarDialog { Text = "Test AutoVar", Object = tbl;
    "First variable:","x", Range(0,4),
    "Second Variable:","y",
    "Domain name:","z", {"one","two","three"; Editable=true},
    "Blonheim's Little Adjustment:","t",
    "Input File:","file",FileIn "Lua (*.lua)|C# (*.cs)",
    "Output File:","outfile",FileOut "Text (*.txt)",
    "Make a Note?","res",
}

if form:ShowDialogOK() then
    print(tbl.x,tbl.z,tbl.res,tbl.file)
end
```

This produces a capable little dialog for editing the table `tbl`'s fields. This style is more often seen with bindings to relational database tables, but fits well with dynamically typed and discoverable data such as a Lua table.

6.2.3 Lua with Objective-C LuaCocoa lets Lua interoperate with Objective-C. There are some cool examples in the documentation.

6.3 I miss all those Python/Perl libraries; can I access them?

LunaticPython allows genuine two way communication between Lua and Python; in particular, you can load Python modules.

Here is a good tutorial on using Lua for Perl programmers.

LuaPerl also allows the calling of Perl code from Lua. As the author says "I have so much Perl code lying around (and there is so much more in the CPAN archives) that it would be a crime to leave this huge potential untapped."

6.4 Can Lua interoperate with CORBA or Web Services?

A comprehensive Lua implementation of CORBA is provided by Oil

There is a commercial product for providing JSON web services with Lua.

It is straightforward to access JSON-style web services:

```
-- Client for the Yahoo Traffic API (http://developer.yahoo.com/traffic/rest/V1/index.html
-- using JSON and Lua
-- Matt Croydon (matt@ooiio.com) http://postneo.com

http = require("socket.http")
json = require("json")

-- Retrieve traffic information for Kansas City, MO
r, c, h = http.request("http://local.yahooapis.com/MapsService/V1/trafficData?appid=LuaDem

if c == 200 then
    -- Process the response
    results = json.decode(r).ResultSet.Result
    -- Iterate over the results
    for i=1,#results do
        print("Result "..i..":")
        table.foreach(results[i], print)
        print()
    end
end
```

See LuaTwitter for another more detailed example of what can be done with Lua and JSON interfaces.

The Kepler Project supports XML-RPC with LuaXMLRPC. Here is a small client example:

```
require "xmlrpc.http"

local ok, res = xmlrpc.http.call ("http://www.oreillynet.com/meerkat/xml-rpc/server.php",
print (ok)
for i, v in pairs(res) do print ('\t', i, v) end
```

7 C API

7.1 How do I traverse a Lua table?

Traversing a general table uses lua_next. This code is the equivalent of using 'pairs' in Lua:

```
/* table is in the stack at index 't' */
lua_pushnil(L);  /* first key */
while (lua_next(L, t) != 0) {
  /* uses 'key' (at index -2) and 'value' (at index -1) */
  printf("%s - %s\n",
         lua_typename(L, lua_type(L, -2)),
         lua_typename(L, lua_type(L, -1)));
  /* removes 'value'; keeps 'key' for next iteration */
  lua_pop(L, 1);
}
```

To go over the elements of an array, use lua_rawgeti. This is the fastest form of array access, since no metamethods will be checked.

For example, this function makes a copy of a table passed as its argument:

```
static int l_copy(lua_State *L) {
    // the table is at 1
    int i,n = lua_objlen(L,1);  // also works for strings, equivalent of #
    lua_createtable(L,n,0); // push our new table, with explicit array size
    for (i = 1; i<=n; i++) {
        lua_rawgeti(L,1,i);
        lua_rawseti(L,-2,i);  // value is at -1, new table is at -2
    }
    return 1; // new table remains on stack
}
```

7.2 How would I save a Lua table or a function for later use?

Newcomers are puzzled that there is no `lua_totable` or `lua_tofunction` corresponding to `lua_tostring`, etc. This is because Lua does not expose a general 'Lua object' type; all values are passed using the stack. The *registry* provides a mechanism for saving references to tables or Lua functions, however.

```
int ref;
lua_newtable(L);  // new table on stack
ref = luaL_ref(L,LUA_REGISTRYINDEX);  // pop and return a reference to the table.
```

Then, when you need to push this table again, you just need to access the registry using this integer reference:

```
lua_rawgeti(L,LUA_REGISTRYINDEX,ref);
```

This also works with functions; you use `luaL_ref` to get a reference to a function which can then be retrieved and called later. This is needed when implementing *callbacks*; remember to use `lua_pcall` so that a bad call does not crash your program.

7.3 How can I create a multi-dimensional table in C?

See this thread on the Lua mailing list. We have a C matrix `V`, which is an array of row arrays, and wish to create the Lua equivalent.

```
int i,j;
lua_createtable(L , nRows, 0);  // push the main table T onto the stack
for (j = 1; j <= nRows; j++ ) {
      lua_createtable(L , nCols, 0); // push a Row Table R onto the stack
      for ( i = 1; i <= nCols; i++ ) {
            lua_pushnumber(L, V[j][i]);
            // value is at -1, R is at -2
            lua_rawseti(L, -2, i);   // R[i] = V[j][i]
      }
      // R is at -1, T is at -2
      lua_rawseti(L, -2, j); // T[j] = R
}
```

7.4 Can I use C++ exceptions?

Lua's basic error mechanism uses `setjmp`, but it is possible to compile Lua so that C++ exceptions can be used.

It is important not to let exceptions propagate across the boundary into Lua, that is, use `try..catch` with your code, and if an error is caught, then use `lua_error` to notify Lua itself.

7.5 How to bind a C++ class to Lua?

Lunar wraps this common pattern:

```
#include "lunar.h"

class Account {
  lua_Number m_balance;
public:
  static const char className[];
  static Lunar<Account>::RegType methods[];

  Account(lua_State *L)       { m_balance = luaL_checknumber(L, 1); }
```

```
  int deposit (lua_State *L) { m_balance += luaL_checknumber(L, 1); return 0; }
  int withdraw(lua_State *L) { m_balance -= luaL_checknumber(L, 1); return 0; }
  int balance (lua_State *L) { lua_pushnumber(L, m_balance); return 1; }
  ~Account() { printf("deleted Account (%p)\n", this); }
};

const char Account::className[] = "Account";

Lunar<Account>::RegType Account::methods[] = {
  LUNAR_DECLARE_METHOD(Account, deposit),
  LUNAR_DECLARE_METHOD(Account, withdraw),
  LUNAR_DECLARE_METHOD(Account, balance),
  {0,0}
};
....
Lunar<Account>::Register(L);
```

This does all the userdata and metatable creation for you; each method of your class iwill be a normal Lua C function that gets its arguments from the stack as usual.

tolua++ is an attractive solution because it can parse 'cleaned' C/C++ headers and generate the bindings.

A very simple example: say we have a burning need for the C function `strstr`; given this file `lib.pkg`:

```
const char *strstr(const char*, const char*);
```

then `tolua -o lib.c lib.pkg` will generate a binding file `lib.c` which can be compiled and linked against the `tolua` library.

`require 'lib'` may result in a surprise, since there is no table `lib` generated, only a global function `strstr`. To put the function into a tble `lib`, `lib.pkg` must look like this:

```
module lib
{
  const char *strstr(const char*, const char*);
}
```

So .pkg files are not really C/C++ headers; however, it is not difficult to convert headers into .pkg files by cleaning them up.

An alternative to tolua is luabind

SWIG is a popular binding generator which can target Lua as well. If a SWIG binding exists for a library, then it is relatively easy to generate Lua bindings from it.

7.6 What's the difference between the lua and the luaL functions?

The `lua_*` functions form the basic API of Lua. It's the communication channel between the C world and the Lua world. It offers the building blocks that are used by the `luaL_*` functions from the auxiliary API to offer some higher level functionality and convenience functions.

7.7 What am I allowed to do with the Lua stack and its contents in my C functions?

The stack you get in your C function was created especially for that function and you can do with it whatever you want. So feel free to remove function arguments that you don't need anymore. However, there is one scenario to watch out for: never remove a Lua string from the stack while you still have a pointer to the C string inside it (obtained with `lua_tostring`):

```
/* assume the stack contains one string argument */
const char* name = lua_tostring(L, 1);
lua_pop(L, 1);  /* careful... */
puts(name);     /* bad! */
```

If you remove the Lua string from the stack there's a chance that you removed the last reference to it and the garbage collector will free it at the next opportunity, which leaves you with a dangling pointer.

7.8 What is the difference between userdata and light userdata?

`lua_newuserdata` allocates a block of memory of the given size. The result is called a `userdatum`, and differs from a block allocated with `malloc` in two important ways: first, it will be collected by the garbage collector, and second its behaviour can specified by a metatable, just like with Lua tables. There are two metamethods which can *only* be used with userdata; `__len` implements the size operator (`#`) and `__gc` provides a function which will be called when the userdatum is garbage collected. A good example of this in the standard Lua library are file types, where `__gc` will close the file handle. The metatable also acts as the unique *type* of a userdatum.

Light userdata, on the other hand, are simple wrappers around a C pointer. They don't have metatables, and aren't garbage-collected. Their purpose is to generate unique 'handles' which can be cheaply compared for equality.

The implementation of a simple array object is discussed in PiL, starting with a simple set of functions and ending up with an object which can be indexed like a regular Lua table.

8 Lua 5.2

The beta version of Lua 5.2 is available from here, currently as 5.2.0-beta. The global variable `_VERSION` will be the string "Lua 5.2", if you need to check for

compatibility. See this wiki page for another catalog, and an in-depth analysis here.

8.1 What Lua 5.1 code breaks in Lua 5.2?

See the official incompatibility list.

8.1.1 Arg for variadic functions The pseudo-variable `arg` in functions with indefinite arguments (`...`) is no longer defined. See here.

8.1.2 Must explicitly require debug library If you need the debug functions, then say `debug = require "debug"`.

8.1.3 unpack moved to table.unpack `unpack` takes a table and returns all its values, e.g. `x,y = unpack {10,20}`. Easy to do a `local unpack = table.unpack` at the top of your files.

8.1.4 setfenv/getfenv deprecated In short, there are no longer any function environments in Lua. There are new ways to handle common usages.

8.1.5 module deprecated This is likely to be the major breaking change, since `module()` is now commonly used when writing Lua modules. module implicitly uses the deprecated `setfenv` and in any case received criticism for causing global namespace pollution.

It is probably best to use a plain 'no magic' style for defining modules, where every exported function is explicitly qualified, and the table of functions explicitly returned.

```
local _M = {}

function _M.one()
   return _M.two()
end

function _M.two()
   return 42
end

return _M
```

Notice that this module does not need to be explicitly named, it may be put anywhere on the module path and required accordingly. But you cannot assume that a global module table has been created, so always use an alias.

```
local mymod = require 'libs.mymod'
```

You can of course use your module name instead of `_M`, this is just to show that the name is arbitrary.

8.1.6 newproxy removed. This was always an 'undocumented' function in Lua 5.1, and it was considered unnecessary, since it was chiefly used to write finalizers. Since the `__gc` metamethod now works for Lua tables, this workaround is no longer needed.

8.2 What are the new Lua 5.2 features?

8.2.1 _ENV and Lexical Scoping The special variable `_ENV` contains the environment for looking up variables. There is no longer anything particularly special about the global table, it is just the default value for `_ENV`.

This little snippet illustrates this special behaviour:

```
-- env1.lua
print(_ENV,_G)
_ENV = { A = 2 }
print 'dolly'
```

And the result is:

```
table: 003D3F08 table: 003D3F08
lua52: env1.lua:3: attempt to call global 'print' (a nil value)
```

In effect, it is just what would happen if you were to change the environment of the current function in Lua 5.1.

This defines a block in which all global accesses happen in a given table:

```
local t = {}
do
  local _ENV = t
  x = 10
  y = 20
end
=> t == {x=10,y=10}
```

The `local` is important here, because we don't wish to modify the effective 'global' environment of the program. `_ENV` is treated like any other local variable.

In an earlier preview version of Lua 5.2 there was an `in t do` syntax that was equivalent to `do local _ENV=t`. However, people found this confusing, perhaps because they were expecting something equivalent to the `with` statement found in Pascal and Visual Basic. The existing global environment is not directly accessible inside the block, so you have to define local variables if the code requires it:

```
local t = {}
local G = _G
do local _ENV = t
  x = 1.2
  y = G.math.sin(x)
end
```

As before,local variables and upvalues are *always* resolved first.

A function may have a modified `_ENV` as an upvalue:

```
local t = {x=10,y=20}
local f1
do local _ENV=t
  function f1()
    return x+y
  end
end

print(f1())
==> 30
```

Note the difference in meaning if `f1` was *not* declared to be local; then `f1` would have become a field of `t`. That is in fact another way to declare modules, much as if you were using `module` without `package.seeall`.

```
local _M = {}
do local _ENV = _M

function f1() return f2() end

function f2() return 42 end

end
return _M
```

The generalized `load` function can compile a chunk of code in a given environment.

```
local fn = load({x=1,y=10},'return x+y')
```

The second argument of `load` may also be a function, which will be called repeatedly to get the chunk text (see load).

To repeatedly call the same function with different environments, it is best to be explicit; set the lexical environment and pass it as a parameter:

```
function(env) local _ENV=env; return x+y end
```

or even more compactly:

```
function (_ENV) return x+y end
```

(see how this is currently done with setfenv.)

8.2.2 Bit Operations Library The functions in `bit32` operate on 32-bit signed integers.

```
> function phex(n) print (('0x%X'):format(n)) end
> phex(bit32.band(0xF0,0x01))
0x0
> phex(bit32.band(0xF0,0xA0))
0xA0
```

There is `band`, 'bshift`,`brotate`,`bor`,`bnot`,`btest`and`bxor'.

This is an important change, since Lua lacks bitwise operators.

8.2.3 Hexadecimal escapes like \xFF allowed in strings Particularly useful when constructing binary strings (remember that the NUL byte is allowed in Lua strings).

```
> s = "\xFF\x01"
> = s:byte(1)
255
> = s:byte(2)
1
```

8.2.4 Frontier Pattern now official To understand why this is useful, you must understand the problem. Consider the common task of searching for whole words in a string. The pattern "%Wdog%W" (i.e. the word surrounded by non-word characters) works in most cases, but fails at the start and beginning of strings, where the respective 'guards' do not match. The frontier pattern `%f[c]` solves the problem: "%f[%w]dog%f[%W]". The first guard says 'match when %w first becomes true' and the second says 'match when %W becomes true', and they both match appropriately at the edges of the strings.

This wiki page is good background reading.

8.2.5 Tables respect ___len metamethod There are a number of metamethods, such as `__gc` and `__len` which only applied to userdata. So there was no way for a pure Lua object to supply its own meaning for the `#` operator, making 'proxy' tables much less useful.

8.2.6 pairs/ipairs metamethods A related problem with Lua objects that want to control their table-like behaviour is that `pairs/ipairs` work directly with a table's raw contents. In Lua 5.2, an object can fully support both kinds of array iteration:

```
for i =1,#obj do ... end  -- can override __len

for i,o in ipairs(obj) do .. end  -- can override __ipairs
```

A more concrete example shows an array proxy object:

```
local array = {}
local ipairs = ipairs
do
  local _ENV = array
  function __len(t)
       return #t.store
  end
  function __newindex(t,i,val)
       t.store[i] = val
  end
  function __index(t,i)
       return t.store[i]
  end
  function __ipairs(t)
       return ipairs(t.store)
  end
end
```

This object will now behave like a regular Lua array-like table:

```
t = new_array()
t[1] = 10
t[2] = 20
print(#t)
for i,v in ipairs(t) do print(i, v) end
=>
2
1       10
2       20
```

To be precise, it acts as a proxy for the underlying object in `store` (which might in fact be userdata).

Array proxies are not as useful as they could be, since the functions in `table` generally work on raw Lua tables. For instance, `table.concat` does not respect `__len` or `__ipairs`.

8.2.7 package.searchpath require looks in both the Lua module and extension path, which are `package.path` and `package.cpath` respectively. Sometimes it is very useful to know where a particular module will be found by Lua, without having to redo the module search logic. `package.searchpath` takes the module name (as understood by `require`) and a string representing the path to be searched:

```
> =  package.searchpath ('lfs',package.cpath)
/usr/local/lib/lua/5.2/lfs.so
> = package.searchpath ('test.mod', package.path)
./test/mod.lua
```

8.2.8 Can get exit status from io.popen `io.popen` is a powerful tool, but often it is useful to get an actual return code as well as the program output. The `close` method will now return the exit status.

```
> f = io.popen 'ls'
> list = f:read '*a'  -- read all the output
> print (#list)
143
> = f:close()
0
```

8.2.9 Yieldable pcall/metamethods Consider writing an iterator which is meant to be used by a coroutine. The iterator function must yield each time, otherwise no other threads can be active.

```
function rand (n)
    return function()
        n = n - 1
        coroutine.yield()
        if n == 0 then return end
        return math.random()
    end
end

function co ()
    for x in rand(10)  do
        print(x)
    end
    return 'finis'
end

f = coroutine.wrap(co)
res = f()
while res ~= 'finis' do
    res = f()
end
```

This little program gives an error with Lua 5.1: "attempt to yield across metamethod/C-call boundary" because the iterator function is being called from C, from the internal implementation of the `for` statement. So people are forced to use an explicit while loop instead, which is awkward.

8.2.10 table.pack `table.pack` is the opposite of `table.unpack`: it takes an indefinite number of values and makes a table. It handles the awkward case of values being `nil` by explicitly setting `n` to the actual table length. So instead of having to say:

```
local arg = {n=select('#',...),...}
```

to handle variable arguments, you can now equivalently say:

```
local arg = table.pack(...)
```

8.2.11 Ephemeron tables From the wikipedia article, "An ephemeron is an object which refers strongly to its contents as long as the ephemeron's key is not garbage collected, and weakly from then on."

In tables with weak keys this means that the key will always be collected when there are no other references to it.

8.2.12 Light C functions Since there are no longer any function environments, a simple reference to a C function is possible, in the same way as light userdata is a simple wrapper around a pointer.

8.2.13 Emergency garbage collection A full garbage collection is forced when allocation fails. This is particularly useful in embedded devices where memory is a constrained resource.

8.2.14 os.execute changes Please note that os.execute now returns a different result; previously it would return the actual return code (usually 0 for success); now it will return `true`, an indication of how the process terminated, and only then the shell return code.

8.2.15 The goto statement Labels look like this `::name::` and you can then say `goto name`. Labels are visible in the block where they are defined. As the manual says "A goto may jump to any visible label as long as it does not enter into the scope of a local variable.".