

Preface

제목 : Programming in Lua, Second Edition

저자 : Roberto Ierusalimschy

Lua는 다른 스크립트 언어와 뭐가 다른가?

Extensibility - C/C++과 인터페이스 쉬워. Fortran, Java, Smalltalk, Ada, C#, Perl, Ruby 과 연동.

Simplicity - 단순한 개념으로 구성. 작고 배우기 쉬워.

Efficiency - 가장 빠른 스크립트 언어 중 하나. 독립 벤치마크에서.

Portability - Playstation, Xbox, Mac OS9, OS X, BeOS, QUALCOMM Brew, MS-DOS, IBM mainframes, RISC OS, Symbian OS, PalmOS, ARM processors, Rabbit processors 및 모든 종류의 Unix, Windows. ANSI C 컴파일러만 있으면 다 돼.

그 밖의 리소스

<http://www.lua.org> 루아 홈

<http://lua-users.org> 사용자 모임

<http://www.inf.puc-rio.br/~roberto/pil2/> 책의 홈

1 Getting Started

2007년 6월 5일 화요일

오후 5:55

전통적인 예제

```
print("Hello World")
```

다음과 같이 실행

```
% lua hello.lua
```

팩토리얼 예제

```
-- defines a factorial function
function fact (n)
    if n == 0 then
        return 1
    else
        return n * fact( n - 1)
    end
end

print("enter a number:")
a = io.read("*number")  -- read a number
print( fact(a))
```

1.1 Chunks

2007년 6월 5일 화요일

오후 6:00

문장 사이를 가리키는 구분자 없음.

아래는 다 똑같아

```
a = 1  
b = a*2
```

```
a = 1;  
b = a*2;
```

```
a = 1; b = a*2  
a = 1 b = a*2 -- ugly, but valid
```

lua 스탠드얼론 인터프리터.

lua 라고만 치면 인터랙트 모드

아래처럼 하면 파일 실행하고 인터랙트 모드로

```
lua -i prog
```

1.2 Some Lexical Conventions

2007년 6월 5일 화요일

오후 6:02

식별자는 영문, 숫자, 언더스코어로 구성. 숫자로 시작하면 안돼.

`_VERSION` 과 같이 하나 혹은 두개의 언더스코어로 시작하는 대문자 식별자는 피해라.

루아에서 쓰고 있다.

로케일에 따라 알파벳외의 문자도 식별자에 포함할 수 있다.

호환성을 생각해서 안쓰는게 좋다.

예약된 단어들

`and break do else elseif end false for function if in local nil not or repeat return
then true until while`

case-sensitive 하다

-- 는 한 줄 주석

다음은 블럭 주석의 예

```
--[[  
print(10)  -- no action (comment)  
--]]
```

아래는 잘못된 블럭 주석 (한 줄 주석이 되어 버린다.)

```
---[[  
print(10)  
--]]
```

1.3 Global Variables

2007년 6월 5일 화요일

오후 6:08

글로벌 변수는 선언하지 않고 그냥 쓰면 된다.

초기화하지 않은 변수에 접근해도 에러 아님, nil 값을 갖는다.

```
print(b) --> nil
```

```
b = 10
```

```
print(b) --> 10
```

따로 해제할 필요는 없지만, 강제로 하고 싶다면

```
b = nil
```

1.4 The Stand-Alone Interpreter

2007년 6월 5일 화요일

오후 6:10

Lua를 사용하는 작은 프로그램

파일 읽을 때 첫줄이 #으로 시작하면 무시한다. shell script 쓰라고

```
#!/usr/local/bin/lua
```

사용법은

```
lua [options] [script [args]]
```

-e 는 커맨드 라인으로 루아 코드를 받는다.

```
% lua -e "print(math.sin(12))" --> -0.53657291800043
```

-i 라이브러리를 로드한다.

```
% lua -i -l a -e "x = 10"
```

```
-- a 라이브러리를 로드하고, x에 10을 대입하고, 인터랙티브 모드로
```

글로벌 변수 _PROMPT 에 값이 있으면 프롬프트로 사용한다.

= 로 시작하는 수식을 넣어서 바로 값을 출력할 수 있다.

```
> = math.sin(3) --> 0.14112000805987
```

```
> a = 30
```

```
> = a --> 30
```

LUA_INIT 환경 변수에 @filename 이 있으면 해당 파일을 실행한다.

LUA_INIT 환경 변수가 @로 시작하지 않으면, 코드라 생각하고 실행한다.

인자를 넣으면 arg 테이블에 인자 정보가 들어간다.

```
% lua -e "sin=math.sin" script a b
```

위와 같이 실행하면, 루아는 아래처럼 인자를 저장한다.

```
arg[-3] = "lua"
```

```
arg[-2] = "-e"
```

```
arg[-1] = "sin=math.sin"
```

```
arg[0] = "script"
```

```
arg[1] = "a"
```

```
arg[2] = "b"
```

... 을 사용해서 전체 인자를 받을 수도. 5.2 절 참조.

2 Types and Values

2007년 6월 5일 화요일

오후 6:23

루아는 동적 타입 언어.

언어상에는 타입 정의가 존재하지 않아
각 값들이 타입을 갖는다.

8 개의 기본 타입

nil, boolean, number, string, userdata, function, thread, table

<code>print(type("Hello world"))</code>	<code>--> string</code>
<code>print(type(10.4*3))</code>	<code>--> number</code>
<code>print(type(print))</code>	<code>--> function</code>
<code>print(type(type))</code>	<code>--> function</code>
<code>print(type(true))</code>	<code>--> boolean</code>
<code>print(type(nil))</code>	<code>--> nil</code>
<code>print(type(type(X)))</code>	<code>--> string</code>

변수는 정해진 타입을 갖지 않는다. 모든 변수는 모든 타입의 값을 포함할 수 있다.

```
print( type( a ) ) --> nil ( 'a' is not initialized )
a = 10
print( type( a ) ) --> number
a = "a string!!"
print( type( a ) ) --> string
a = print      -- yes, this is valid!
a( type( a ) ) --> function
```

2.1 Nil

2007년 6월 5일 화요일

오후 6:32

하나의 값(nil)을 갖는 타입.

글로벌 변수의 기본값은 nil

글로벌 변수를 해제하기 위해 nil을 대입할 수 있다.

2.2 Booleans

2007년 6월 5일 화요일

오후 6:33

true, false 값을 갖는다.

조건문의 결과값으로 사용된다.

 false, nil -> false

 그 밖의 모든 값 -> true

다른 언어와 달리 0과 빈스트링도 true가 된다.

2.3 Numbers

2007년 6월 5일 화요일

오후 6:34

real number (double) 만 있다.

사람들은 부동소수점 연산 에러에 대한 오해를 가지고 있다.

단순한 증가 연산도 걱정한다.

하지만 10의 14승이 넘지 않는다면, 부동소수점으로 정수를 표현하는데 아무런 문제가 없다.

루아 넘버는 32bit 정수를 라운딩 문제 없이 표현할 수 있다.

요즘의 CPU는 정수 연산보다 실수 연산이 빠르거나 비슷하다.

다음과 같이 표현 가능

4 0.4 4.57e-3 0.3e12 5e+20

2.4 Strings

2007년 6월 7일 목요일

오전 10:55

8 bit clean

UTF8이나 ISO8859 시리즈의 문자열을 다룰 수 있다.

immutable values

수정이 필요한 경우 새 문자열을 생성한다.

```
a = "one string"
b = string.gsub( a, "one", "another" ) -- change string parts
print( a ) --> one string
print( b ) --> another string
```

스트링은 메모리 관리 대상

루아는 큰 스트링도 효율적으로 다룸

100K 나 1M 문자열을 다루는 것은 보통의 일.

더블 쿼츠나 싱글 쿼츠로 표현

```
a = "a line"
b = 'another line'
```

C 와 비슷한 이스케이프 시퀀스를 가진다.

double square bracket 으로도 긴 문자열을 나타낼 수 있다.

```
page = [[
<html>
....
</html>
]]
```

문자열 안에 [[가 있는 경우를 대비해서 [=,]=] 쌍으로도 문자열을 표현할 수 있다.

=의 개수를 맞추면 된다.

이런 특징은 주석에도 해당

루아는 문자열과 숫자 사이에 자동 형변환

문자열에 가해진 numeric 연산은 스트링을 숫자로 바꾼다.

```
print("10" + 1 ) --> 11
```

숫자에 가해진 문자열 연산의 경우 그 반대로..

```
print( 10 .. 20 ) --> 1020
```

이게 좋은 아이디어인지 확신할 수 없다. 이 기능에 의지하지 말자.

이런 기능에도 불구하고 숫자와 문자열은 다른 것

10 = "10" 은 false 다.

tonumber(문자열) 로 숫자로 변환

tostring(숫자) 로 문자로 변경

연산자. 문자열의 길이를 반환한다.

```
a = "hello"
print( #a ) --> 5
print("#goodW0bye") --8
-- 널문자에 상관없이 길이를 올바르게 계산한다.
```

2.5 Tables

2007년 6월 7일 목요일

오전 11:17

associative arrays 를 구현한 것.

= 숫자 뿐 아니라 문자열 혹은 다른 어떤 값을 인덱스로 쓸 수 있다. nil은 빠고.

고정된 사이즈 없음. 동적으로 조정.

루아의 메인 (사실, 유일한) data structuring mechanism.

보통의 배열, 심볼 테이블, 셋, 레고드, 큐 그리고 그밖의 다른 자료 구조를 표현하는 데 사용
간단하고, 일원화되어 있고, 효율적인 방법

루아는 모듈, 패키지, 객체를 나타내는 데도 테이블을 사용.

io.read -> io 모듈의 read 함수

실상은, "read"를 키로 사용해서 테이블 "io"를 검색해라.

Table은 값도 변수도 아닌, 객체!

{ } 를 사용해서 테이블 생성

```
a = {} -- create a table and store its reference in 'a'
k = "x"
a[k] = 10 -- new entry, with key="x" and value=10
a[20] = "great" -- new entry, with key=20 and value="great"
print( a["x"] ) --> 10
k = 20
print(a[k]) --> "great"
a["x"] = a["x"] + 1 -- increments entry "x"
print(a["x"]) --> 11
```

테이블은 익명, 테이블을 담고 있는 변수와 테이블은 아무런 관계가 없다.

```
a = {}
a["x"] = 10
b = a -- 'b' refers to the same table as 'a'
print(b["x"]) --> 10
b["x"] = 20
print(a["x"]) --> 20
a = nil -- only 'b' still refers to the table
b = nil -- no references left to the table
```

각 테이블은 다른 타입의 인덱스로 값을 저장할 수 있다. 그리고 새 엔트리를 수용하기 위해서 확장한다.

```
a = {} -- empty table
-- create 1000 new entries
for i = 1, 1000 do a[i] = i*2 end
print(a[9]) --> 18
a["x"] = 10
print(a["x"]) --> 10
print(a["y"]) --> nil
```

초기화 되지 않은 필드에 대해서 nil을 반환.

초기화 되지 않은 글로벌 변수도 nil을 반환했음.

우연이 아니라, 글로벌 변수도 테이블에 보관하기 때문.

a.name 의 표현도 가능

```
a.x = 10 -- same as a["x"] = 10
print( a.x ) -- same as print(a["x"])
print(a.y) -- same as print(a["y"])
```

a.x 를 a[x]와 헛갈리지 말자. a["x"]가 맞다.

```
a = {}
x = "y"
a[x] = 10      -- put 10 in field "y"
print(a[x]) --> 10 -- value of field "y"
print(a.x) --> nil -- value of field "x" (undefined)
print(a.y) --> 10 -- value of field "y"
```

보통의 배열이나 리스트로 표현하려면, 그냥 정수를 키로 사용하면 된다.

```
크기를 지정할 방법도 없고, 필요도 없다.
-- read 10 lines storing them in a table
a = {}
for i = 1,10 do
    a[i] = io.read()
end
```

루아에서 배열이 시작은 1 (0이 아니다)!!!

연산자는 배열이나 리스트의 마지막 인덱스 (혹은 크기라고 볼 수도 있다)를 반환

```
-- print the lines
for i = 1, #a do
    print(a[i])
end
```

연산자는 다음과 같은 관용구를 만들어 낸다.

```
print (a[#a]) -- prints the last value of list 'a'
a[#a] = nil -- removes this last value
a[#a+1] = v -- appends 'v' to the end of the list
```

연산자가 배열의 끝을 찾는 방법은?

1 번째 필드부터 시작해서 마지막 nil을 찾는다.

배열의 중간에 nil이 있으면 올바르게 동작하지 않는다.

table.maxn() -> 가장 큰 양수의 인덱스를 반환

```
a = {}
a[10000] = 1
print(table.maxn(a)) --> 10000
```

2.6 Functions

2007년 6월 7일 목요일

오전 11:42

루아에서 함수는 1등급 값.

함수가 변수에 보관될 수 있고, 인자로 넘겨질 수 있고, 반환값으로서 반환될 수 있다는 뜻.

뒤에서 자세히.

루아에서 C로 만든 함수를 호출할 수 있다.

루아의 모든 표준 라이브러리는 C로 작성되었다.

문자열 다루기, 테이블 다루기, I/O, 운영체제에 접근하기, 수학 함수들, 디버깅 함수 등.

2.7 Userdata and Thread

2007년 6월 7일 목요일

오전 11:45

유저데이터 타입이 있어서 Lua 변수에 임의의 C 데이터를 보관할 수 있다.

대입과 동등비교 외에 다른 연산은 할 수 없다.

C로 작성된 라이브러리나 어플리케이션에 의해 생성된 새로운 타입을 Lua 에서 쓰기 위한 용도.

Thread는 9장에서.

3 Expressions

2007년 6월 7일 목요일

오후 1:46

냉무

3.1 Arithmetic Operators

2007년 6월 7일 목요일

오후 1:47

다음의 산술연산 지원

- + addition
- Subtraction
- * multiplication
- / division
- ^ exponentiation
- % modulo (5.1 에 추가된 것)
- negation

모든 연산자는 실수를 받음

`x^0.5` --> square root of x

소수점 2자리까지만 자르는 예

`x = math.pi`

`print(x - x%0.01)` --> 3.14

3.2 Relational Operators

2007년 6월 7일 목요일

오후 1:52

아래 연산자 제공

```
< > <= >= == ~=
```

타입이 다른 값들은 항상 같지 않다.

테이블, 유저데이터, 함수의 경우 reference만 비교한다.

```
a = {}; a.x = 1; a.y = 0
```

```
b = {}; b.x = 1; b.y = 0
```

```
c = a
```

```
-- a== c but a ~= b
```

문자열 비교시 알파벳순, 세팅된 로케일을 따른다.

다른 타입의 관계연산은 에러

```
2 < "15"
```

3.3 Logical Operators

2007년 6월 7일 목요일

오후 1:55

다음의 연산자가 있다

and or not

false 와 nil 은 false로 취급

그밖의 다른 값은 true

and 연산자의 행동방식

1. 첫번째 연산자가 false면 첫 번째 연산자 반환
2. 그렇지 않으면 두 번째 연산자 반환

or 연산자의 행동방식

1. 첫번째 연산자가 false가 아니면 첫번째 연산자 반환
2. 그렇지 않으면 두 번째 연산자 반환

and or 연산자 사용 예

```
print(4 and 5) --> 5
print(nil and 13) --> nil
print(false and 13) --> false
print(4 or 5) --> 4
print(false or 5) --> 5
```

and or 연산자는 short-cut evaluation

필요한 경우만 두 번째 연산자를 참조한다.

C에서 $a ? b : c$ 와 같이 하려면

```
max = ( x > y ) and x or y
```

3.4 Concatenation

2007년 6월 7일 목요일

오후 2:01

문자열 연결

..

문자열은 불변의(immutable) 값이므로, 연결 연산자는 새 문자열을 만들어낸다.

```
a = "Hello"
```

```
print(a .. " World") --> Hello World
```

```
print(a)             --> Hello
```

3.5 Precedence

2007년 6월 7일 목요일

오후 2:04

다음과 같다

^

not # - (unary)

* / %

+ -

..

< > <= >= ~= ==

and

or

^ 와 .. 을 제외하고 모든 바이너리 연산자는 왼쪽먼저 연산

3.6 Table Constructors

2007년 6월 7일 목요일

오후 2:06

테이블을 만들고 초기화하는 표현.

빈 테이블을 만드려면 {} 로.

배열을 초기화 할 수도 있다.

```
days = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
        "Saturday"}
```

첫번째 인덱스는 1이 된다.

```
print(days[4]) --> Wednesday
```

다음과 같은 초기화도 가능

```
a = {x=10, y=20}
```

아래와 같은 의미

```
a = {}; a.x=10; a.y=20
```

어떤 방식으로 생성하건 상관없이, 언제라도 필드를 추가하고 삭제할 수 있다.

콘솔로 입력받은 값으로 링크드리스트를 만드는 예

```
list = nil
for line in io.lines() do
    list = {next=list, value=line}
end
```

하나의 생성자 안에서 여러가지 초기화 방식을 섞어서 사용할 수 있다.

```
polyline = {color="blue", thickness=2, npoints=4,
            {x=0, y=0},
            {x=-10, y=0},
            {x=-10, y=1},
            {x=0, y=1}
            }
print(polyline[2].x) --> -10
print(polyline[4].y) --> 1
```

이런 초기화 방식의 단점은 음수의 인덱스나 적절하지 않은 문자를 포함한 인덱스를 만들 수 없다. 그래서 이런 문법이 있다.

```
opnames = { ["+"] = "add", ["-"] = "sub",
            ["*"] = "mul", ["/"] = "div" }
i = 20; s = "-"
a = {[i+0] = s, [i+1] = s..s, [i+2] = s..s..s}
print(opnames[s]) --> sub
print(a[22]) --> ---
```

인덱스가 0부터 시작하게 하려면 이렇게

```
days = { [0]="Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
        "Saturday"}
```

0이 다른 필드에 영향을 미치는 것은 아님.

이런 방식 비추.

1부터 시작한다고 가정하는 내장 함수들과 호환되지 않으므로.

초기화 리스트에서 , 대신에 ;를 사용해도 좋다

```
{x=10, y=45; "one", "two", "three"}
```

4 Statements

2007년 6월 7일 목요일

오후 5:26

C 나 파스칼과 비슷한 전형적인 세트의 명령문을 지원

4.1 Assignment

2007년 6월 7일 목요일

오후 5:27

기본적인 할당

```
a = "hello" .. "world"
```

```
t.n = t.n + 1
```

다중 할당이 된다.

```
a, b = 10, 2*x
```

모든 값을 읽은 후에 할당이 되므로, swap 작업에 써도 좋다.

```
x, y = y, x
```

```
a[i], a[j] = a[j], a[i]
```

대입할 값이 모자르면 nil을, 값이 남으면 무시.

```
a, b, c = 0, 1
```

```
print(a, b, c) --> 0 1 nil
```

```
a, b = a+1, b+1, b+2 -- value of b+2 is ignored
```

```
print(a, b) --> 1 2
```

```
a, b, c = 0
```

```
print(a, b, c) --> 0 nil nil
```

다중 할당은 그다지 자주 쓸 일은 없다.

여러번의 할당보다 빠르지 않다.

그러나 swap 이나 다중 반환값의 용도로 좋다.

```
a, b = f()
```

4.2 Local variables and Blocks

2007년 6월 7일 목요일

오후 5:31

로컬 변수는 이렇게

```
j = 10 -- global variable
local i = 1 -- local variable
```

로컬 변수는 자신이 정의된 블록 안쪽으로 범위가 제한

블록이란 제어문의 바디, 함수의 바디, 혹은 청크

청크는 변수가 정의된 파일이나 스트링

```
x = 10
local i = 1 -- local to the chunk
while i <= x do
    local x = i * 2 -- local to the chunk
    print(x) --> 2, 4, 6, 8, ...
    i = i + 1
end
```

팁. 위의 예제를 인터랙티브 모드에서 입력할 때는 do end 로 묶어줘야 하나의 청크로 만들 수 있다.

가능하다면 글로벌보다 로컬 변수를 쓰는 것이 좋은 습관

더 빠르다.

초기값이 없다면 nil이 들어간다.

4.3 Control Structures

2007년 6월 7일 목요일

오후 5:37

복습. 루아는 false와 nil을 제외한 모든 값을 true로 취급. 0조차도.

```
if then else
  if a <0 then a = 0 end
  if op == "+" then
    r = a + b
  elseif op == "-" then
    r = a - b
  else
    error("invalid operation")
  end
end
while
  local i = 1
  while a[i] do
    print(a[i])
    i = i + 1
  end
end
repeat
  -- print the first non-empty input line
  repeat
    line = os.read()
  until line ~= ""
  print(line)
```

다른 언어들과 달리, 루프 안에서 선언한 로컬 변수를 조건식 안에서도 접근 가능
(5.1에서 소개된 것)

```
local sqr = x/2
repeat
  sqr = (sqr + x/sqr)/2
  local error = math.abs(sqr^2 - x)
until error < x/10000 -- 'error' still visible here
```

numeric for

다음과 같이 쓴다. step 값은 옵션으로 +1이 디폴트.

```
for i = 1, f(x) do print(i) end
```

```
for i=10,1,-1 do print(i) end
```

for 의 인자들은 처음 한 번만 해석된다.

for 문에서 정의된 컨트롤 변수는 루프에 local 이 된다.

```
for i =1,10 do print(i) end
```

```
max = i -- probably wrong! 'i' here is global
```

break 를 사용해서 빠져나올 수도 있다.

Generic for

iterator 함수에서 반환하는 모든 값을 돈다.

```
-- print all values of array 'a'
```

```
for i, v in ipairs(a) do print(v) end
```

```
-- print all keys of table 't'
```

```
for k in pairs(t) do print(k) end
```

4.4 break and return

2007년 6월 8일 금요일

오후 6:26

break와 return 은 블록 밖으로 나가게 해준다.

break와 return 은 블록의 마지막 문장으로서만 나올 수 있다

디버깅 용도로 문장 중간에 넣으려면 아래처럼 할 수도

```
function foo()
  return --<< SYNTAX ERROR
  -- 'return' is the last statement in the next block
do return end -- OK
  <other statements>
end
```

5 Functions

2007년 6월 8일 금요일

오후 6:29

함수호출에 인자가 없어도 () 는 써줘야한다.

하나의 인자만 있고, 그 인자가 문자열이나 테이블 생성자라면 괄호는 생략 가능

```
print "Hello World" <--> print("Hello World")
f{x=10, y=20} <--> f({x=10, y=20})
```

객체지형적인 호출을 위한 콜론 연산자.

```
o:foo(x) <--> o.foo(o,x)
```

함수를 정의하는 예

```
function add (a)
  local sum = 0
  for i, v in ipairs(a) do
    sum = sum + v
  end
  return sum
end
```

매개변수는 로컬 변수와 같이 동작한다.

인자의 개수가 매개변수보다 많으면 남는 것 무시, 인자의 개수가 적으면 nil이 대입

```
function f(a, b) return a or b end
f(3) -- a=3, b=nil
f(3, 4) -- a=3, b=4
f(3, 4, 5) -- a=3, b=4 ( 5 is discarded)
```

위의 행동이 런타임 에러를 내기 쉽지만 유용하다. 특히 디폴트 값 용으로.

```
function incCount(n)
  n = n or 1
  count = count + n
end
```

5.1 Multiple Results

2007년 6월 8일 금요일

오후 6:41

루아 함수는 다수의 결과를 반환할 수 있다.

예를 들어 `string.find` 함수는 결과의 시작 인덱스와 끝 인덱스를 반환한다.

```
s, e = string.find("hello Lua users", "Lua")
print(s, e) --> 7 9
```

배열에서 가장 큰 값과 그 값의 위치를 반환하는 예

```
function maximum (a)
    local mi = 1 -- index of the maximum value
    local m = a[mi] -- maximum value
    for i, val in ipairs(a) do
        if val > m then
            mi = i; m = val
        end
    end
    return m, mi
end
print( maximum({8,10,23,12,5})) --> 23 3
```

함수를 호출한 문맥에 맞게 반환값을 조절한다.

문장(statement)으로서 호출한 경우 반환값을 없앤다

표현(expression)으로서 호출한 경우 첫번째 반환값만 유지한다.

리스트의 마지막 표현인 경우 모든 반환값이 유지된다.

반환값이 조절되는 예

```
function foo0 () end -- returns no results
function foo1() return "a" end -- returns 1 result
function foo2() return "a", "b" end -- returns 2 results
```

```
x, y = foo2() -- x="a", y="b"
x = foo2() -- x="a", "b" is discarded
x,y,z = 10, foo2() -- x=10, y="a", z="b"
```

```
x, y = foo0() -- x=nil, y=nil
x,y = foo1() -- x="a", y=nil
x,y,z = foo2() -- x="a", y="b", z=nil
```

```
x,y = foo2(), 20 -- x="a", y=20
x,y = foo0(), 20, 30 -- x=nil, y=20, 30 is discarded
```

```
print(foo0()) -->
print(foo1()) --> a
print(foo2()) --> a b
print(foo2(), 1) --> a 1
print(foo2() .. "x") --> ax
```

```
t = {foo0()} -- t = {} (an empty table)
t = {foo1()} -- t = {"a"}
t = {foo2()} -- t = {"a", "b"}
```

```
t = {foo0(), foo2(), 4} -- t[1] = nil, t[2] = "a", t[3] = 4
```

```
print((foo0())) --> nil
```

```
print((foo1())) --> a
```

```
print((foo2())) --> a (only 1 result by parentheses )
```

테이블의 모든 원소를 반환하는 방법

```
print(unpack{10,20,30}) --> 10 20 30
```

```
a,b = unpack{10,20,30} -- a=10, b=20, 30 is discarded
```

generic call에 유용하다.

5.2 Variable Number of Arguments

2007년 6월 11일 월요일

오후 2:21

모든 인자의 합을 반환하는 함수

```
function add(...)
  local s = 0
  for i, v in ipairs{...} do
    s = s + v
  end
  return s
end
print(add(3, 4, 10, 25, 12)) --> 54
```

다중 반환값처럼 동작한다.

```
local a, b = ...
```

foo() 함수를 감싸서 로깅을 하는 foo1() 함수의 예

```
function foo1(...)
  print("calling foo:", ...)
  return foo(...)
end
```

가변 인자 탐색하기

```
for i=1, select('#', ... ) do
  local arg = select( i, ... ) -- get i-th parameter
  <loop body>
end
```

5.3 Named Arguments

2007년 6월 11일 월요일

오후 2:27

루아는 위치를 사용해서 인자와 매개변수를 맞춘다.

루아에 매개변수에 이름을 줄 수 있는 방법은 없다!!!

테이블을 사용해서 동일한 결과를 얻을 수는 있다.

```
w = Window{ x=0, y=0, width=300, height=200, title="Lua", background="blue" }
```

```
function Window (options)
    _Window( options.title, options.x or 0, 기타 생략 )
end
```

6 More about Functions

2007년 6월 11일 월요일

오후 2:58

루아에서 함수는 'first class values'

변수와 테이블에 저장 가능, 인자로 넘길 수도, 반환값으로 쓰일 수도 있다.

루아에서 함수는 익명이다. 다만 이름이 있는 변수에 보관되고 있을 뿐이다.

```
a = { p = print }
a.p("Hello World") --> Hello World
print = math.sin -- 'print' now refers to the sine function
a.p(print(1))      -- 0.841470
sin = a.p          -- 'sin' now refers to the print function
sin(10,20)        --> 10 20
```

다음의 두 줄은 완전히 똑같다.

```
function foo (x) return 2*x end
foo = function (x) reutrn 2*x end
```

Higher-order functions

함수를 인자로 받는 함수.

table 유틸리티 함수인 sort의 경우 비교 기준을 함수로 받는다.

```
network = {
  { name = "grauna", IP = "210.26.30.34" },
  중간 생략
}
table.sort( network, function (a, b) return (a.name > b.name) end )
```

6.1 Closure

2007년 6월 11일 월요일

오후 4:08

lexical scoping

함수 A가 함수 B 안에서 작성된 경우, B의 모든 지역 변수에 접근할 수 있다.

이런 것이 가능하다.

```
function newCounter()
  local i = 0
  return function () -- anonymous function
    i = i + 1
    return i
  end
end
```

end

```
c1 = newCounter()
```

```
print(c1()) --> 1
```

```
print(c1()) --> 2
```

c1을 호출하는 시점에서 newCounter() 함수는 끝난 상태. 하지만 Closure라는 개념을 사용해서 이런 상황도 잘 처리한다.

Closure

클로저란 함수 + 함수가 접근하는 모든 non-local 변수들. (upvalues)

newCounter() 함수를 다시 호출하면, 새로운 지역변수 i를 사용해서 새로운 익명 함수가 생성된다.

```
c2 = newCounter()
```

```
print(c2()) --> 1
```

```
print(c1()) --> 3
```

```
print(c2()) --> 2
```

클로저의 장점을 활용한 예. 계산기 예제의 숫자 버튼. 10개의 비슷한 버튼을 만드는 대신 아래와 같이.

```
function digitButton (digit)
  return Button{ label = tostring(digit),
    action = function()
      add_to_display(digit)
    end
  }
end
```

end

클로저 활용의 또 다른 예.

기본적인 sin 함수의 재정의

```
oldSin = math.sin
```

```
math.sin = function (x)
```

```
  return oldSin(x*math.pi/180)
```

```
end
```

이렇게 하면 누군가 oldSin 의 값을 바꿀 수 있으니, 이렇게 숨길 수 있다.

```
do
```

```
  local oldSin = math.sin
```

```
  local k = math.pi/180
```

```
  math.sin = function (x)
```

```
    return oldSin(x*k)
```

```
  end
```

```
end
```

6.2 Non-Global Functions

2007년 6월 11일 월요일

오후 5:06

함수는 테이블에 저장 가능

로컬 변수에도 저장 가능 (로컬 함수처럼 된다.)

```
local f = function ( <params> )
  <body>
end
```

다음과 같은 편리한 문법도 있다. syntactic sugar

```
local function f ( <params> )
  <body>
end
```

다음의 실수 주의

```
local fact = function (n)
  if n == 0 then return 1
  else return n*fact(n-1) -- buggy
end
end
```

글로벌 함수 fact()가 참조될 수 있다.

아래와 같이 고치자. (사실상 syntactic sugar 가 이렇게 된다.)

```
local fact
fact = function (n )
  이하 같음.
```

6.3 Proper Tail Calls

2007년 6월 11일 월요일

오후 5:12

tail call은 함수의 마지막에서 다른 함수를 호출할 때 일어난다. 아래에서 g에대한 호출은 tail call

```
function f() return g(x) end
```

f에서 더이상 할 일이 없으므로, 스택에서 f와 관련한 정보를 제거한다. g()가 끝나면 f()를 호출한 곳으로 바로 반환.

다음과 같은 재귀 함수도 절대로 스택 오버플로우가 나지 않는다.

```
function foo (n)
  if n > 0 then return foo(n - 1) end
end
```

tail call 인듯 하지만 아닌 경우의 예

```
function f(x) g(x) end -- f() should discard results from g()
return g(x) + 1      -- must do the addition
return x or g(x)     -- must adjust to 1 result
return (g(x))        -- must adjust to 1 result
```

tail call 아닌듯 하지만 맞는 예

```
return x[i].foo(x[j] + a*b, i + j)
```

7 Iterators and the Generic for

2007년 6월 11일 월요일

오후 5:19

省略

8 Compilation, Execution, and Errors

2007년 6월 13일 수요일

오후 3:08

인터프리터 언어지만, 실행 전에 중간 형태로 프리 컴파일을 한다.

8.1 Compilation

2007년 6월 13일 수요일

오후 3:09

dofile() 은 루아 파일을 실행, loadfile()은 루아 파일을 로드. loadfile() 은 에러 안냄.

```
function dofile (filename)
    local f = assert(loadfile(filename))
    return f()
end
```

loadstring()은 문자열 명령을 로드.

```
f = loadstring("i = i + 1")
i = 0
f(); print(i) --> 1
f(); print(i) --> 2
```

다음은 두 줄은 비슷하다고 볼 수 있지만, 아래 쪽이 더 빠르다. 추가적인 컴파일을 요하지 않기 때문.

```
f = loadstring("i = i + 1")
f = function() i = i + 1 end
```

또한 loadstring() 은 lexical scoping 을 염두하지 않으므로, 다음과 같은 차이를 갖는다.

```
i = 32
local i = 0
f = loadstring("i = i + 1; print(i)")
g = function () i = i + 1; print(i) end
f() --> 33
g() --> 1
```

일반적인 실수. 아래 예에서 루아 파일을 로딩한다고 foo() 함수가 정의되는 것은 아니다.

```
-- foo.lua
function foo(x)
    print(x)
end
```

```
f = loadfile("foo.lua")
```

```
print(foo) --> nil
f() --> defines 'foo'
foo("ok") --> ok
```

8.2 C Code

2007년 6월 13일 수요일

오후 3:30

dll등의 동적 모듈을 로딩하는 ANSI C 표준은 없지만, 지원하기로 함.

package.loadlib() 로 로드

```
local path = "/usr/local/lib/lua/5.1/socket.so"  
local f = package.loadlib(path, "luaopen_socket")
```

8.3 Errors

2007년 6월 13일 수요일

오후 3:33

루아는 어플리케이션 안에 내장되는 확장 언어의 용도가 크므로, 에러가 발생하면 비정상 종료하는 대신 현재 체크를 끝내고 어플리케이션에 반환한다.

숫자가 아닌 두 값을 더하려고 할때나, 함수가 아닌 값을 호출하려고 할때 등등 에러가 발생한다.

`error()` 함수로 강제로 일으킬 수도

```
print "enter a number:"  
n = io.read("*number")  
if not n then error("invalid input") end
```

위와 같은 구문이 자주 쓰이므로, `assert` 를 제공한다.

```
print "enter a nuber:"  
n = assert( io.read("*number", "invalid input"))
```

8.3 Error Handling and Exceptions

2007년 6월 13일 수요일

오후 3:38

대부분의 경우 루아에서 에러 핸들링 할 필요는 없다. 어플리케이션에서 하게 된다.

루아에서 에러를 핸들링할 필요가 있다면, pcall(protected call)을 쓰서 코드를 감싸야 한다.

```
function foo()
  <some code>
  if unexpected_condition then error() end
  <some code>
  print(a[i]) -- potential error: 'a' may not be a table
  <some code>
end
```

```
if pcall(foo) then
  -- no errors while running 'foo'
  <regular code>
else
  -- 'foo' raised an error: take appropriate actions
  <error-handling code>
end
```

pcall()로 호출한 함수 안에서 에러를 받는 법

```
local status, err = pcall( function() error( {code=121} ) end )
print(err.code) --> 121
```

xpcall() 은 pcall() 과 같지만 에러 핸들링 함수를 더 받는다.

함수에서 에러가 발생하고 스택이 언와인딩되기 전에 에러 핸들링 함수가 호출된다.

debug.debug, debug.traceback 등의 함수로 자세한 정보를 얻는다.

9 Coroutines

2007년 6월 13일 수요일

오후 4:56

생략

10 Complete Examples

2007년 6월 13일 수요일

오후 4:58

11 Data Structures

2007년 6월 13일 수요일

오후 4:58

생략

12 Data Files and Persistence

2007년 6월 13일 수요일

오후 4:58

데이터 파일을 쓰는 것보다 읽는 것이 어려움.

루아를 통해 데이터를 읽는 코드를 전혀 생성하지 않고도 읽을 수 있는 법을 알아보자

12.1 Data Files

2007년 6월 13일 수요일

오후 5:17

데이터 파일을 이렇게 만들면

```
Entry {
  author = "Donald E. Knuth",
  title = "Literate Programming",
  publisher = "CSLI",
  year = 1992
}
Entry {
  author = "Jon Bentley"
  title = "More Programming Pearls",
  year = 1990,
  publisher = "Addison-Wesley",
}
```

저자들의 이름을 출력하는 코드는 이렇게 된다.

```
local authors = {} -- a set to collect authors
function Entry (b) authors[b.author] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

루아는 실행뿐만 아니라 컴파일도 빠르다.

위 코드는 2Mbytes의 데이터를 읽어서 처리하는데 1초 이하가 걸린다.

나머지는 생략

13 Metatables and Metamethods

2007년 6월 13일 수요일

오후 5:31

생략

14 The Environment

2007년 6월 13일 수요일

오후 5:31

루아는 글로벌 변수들을 `environment` 라 불리는 보통의 테이블에 보관한다.

그래서 루아의 내부 구현이 간단해졌다.

보통의 테이블처럼 다룰 수 있다.

`_G` 라는 글로벌 변수에 이 테이블을 보관한다.

`_G._G` 는 `_G` 와 같다.

모든 글로벌 변수의 이름을 출력해보자

```
for n in pairs(_G) do print(n) end
```

14.1 Global Variables with Dynamic Names

2007년 6월 13일 수요일

오후 5:47

변수 `varname` 에 글로벌 변수의 이름이 담겨 있다면, 다음과 같이 그 값을 구할 수 있다.

```
value = _G[varname]
```

다음의 두 줄은 같은 코드다

```
_G["a"] = _G["var1"]
```

```
a = var1
```

14.2 Global-Variable Declarations

2007년 6월 13일 수요일

오후 5:53

생략, metadata와 관련한 부분

15 Modules and Packages

2007년 6월 13일 수요일

오후 5:54

생략

16 Object-Oriented Programming

2007년 6월 13일 수요일

오후 5:57

17 Weak Tables

2007년 6월 13일 수요일

오후 5:57

생략

Part III The Standard Libraries

2007년 6월 13일 수요일

오후 6:04

- 18 The Mathematical Library
- 19 The Table Library
- 20 The String Library
- 21 The I/O Library
- 22 The Operating System Library
- 23 The Debug Library

24 An Overview of the C API

2007년 6월 13일 수요일

오후 6:06

C API 는 C 코드가 루아와 통신하게 해주는 함수들의 모음

루아의 글로벌 변수를 읽고 쓰기, 루아 함수 호출하기, 루아 코드 조각 실행하기, 루아에서 부를 수 있는 C 함수 등록하기 등의 기능.

루아와 C사이의 통신의 주요 수단은 가상의 '스택'이다.

거의 모든 API 호출은 이 스택 상의 값에 대해서 작업을 한다.

중간단계의 결과를 보관하기 위해서도 스택을 쓴다

스택은 루아의 C의 두 가지 중요한 불일치를 해결해준다.

루아는 GC 기반, C는 명시적인 메모리 해제.

루아는 동적 타이핑, C는 정적 타이핑.

24.1 A First Example

2007년 6월 13일 수요일

오후 6:08

간단한 stand-alone 루아 인터프리터

```
#include <stdio.h>
#include "lua.h"
#include "luaolib.h"
#include "lualib.h"

int main (void) {
    char buff[256];
    int error;
    lua_State *L = luaL_newstate(); /* opens Lua */
    luaL_openlibs(L); /* opens the standard libraries */

    while (fgets(buff, sizeof(buff), stdin) != NULL ) {
        error = luaL_loadbuffer(L, buff, strlen(buff), "line") ||
            lua_pcall(L, 0, 0, 0);
        if (error) {
            fprintf(stderr, "%s", lua_tostring(L, -1));
            lua_pop(L, 1); /* pop error message from the stack */
        }
    }

    lua_close(L);
    return 0;
}
```

lua.h

루아의 기본 기능 제공

새 루아 환경을 생성, 루아 함수를 호출, 전역 변수를 읽고 쓰기, C 함수 등록 등.

luaolib.h

보조 라이브러리

여기 있는 함수들은 luaL_ 의 프리픽스를 갖는다.

lua.h 에 정의한 함수를 사용해서 구현됨.(Lua 내부에 대한 접근은 없음)

lua_State

루아 라이브러리에는 전역 변수가 전혀 없음

모든 상태는 lua_State에 보관, 이 녀석이 API들에 전달됨.

lua_newstate

새로운 루아 환경을 생성

처음 생성되었을 때는 기본 라이브러리조차 로딩되지 않은 상태 (print도 안됨)

루아를 가볍게 유지하기 위해

luaL_openlibs

모든 표준 라이브러리를 연다.

luaL_loadbuffer

루아 코드를 컴파일

에러가 없으면 0을 반환, 컴파일 결과 생성된 청크를 스택에 보관한다.

lua_pcall

스택에서 청크를 꺼내서 프로텍티드 모드로 실행한다.

lua_tostring

luaL_loadbuffer, lua_pcall은 실패한 경우 에러메시지를 스택에 넣어둔다.

lua_tostring을 사용해서 얻어낼 수 있다.

lua_pop

스택에 있는 에러메시지를 꺼내온다.

루아 C API 들은 C 언어를 가정하고 있기 때문에, C++ 컴파일러에서 돌리려면 extern "C" 안에 #include 해야 한다.

24.2 The Stack

2007년 6월 14일 목요일

오전 10:40

루아의 동적 타입을 C로 표현하려면 어렵다.

$a[k] = v$ 를 표현하는 C 함수를 상상해보자면(lua_Value 는 유니온 타입)

```
void lua_settable( lua_Value a, lua_Value k, lua_Value v );
```

단점 1. 이런 복잡한 타입을 다른 언어에 적용하는 건 어렵다.

단점 2. 루아는 GC를 한다. C 변수에 대해서는 어찌할 도리가 없다.

그래서 Stack 을 통해서 값을 주고 받게 했다.

여전에 Stack에 값을 넣거나 빼는 함수는 '타입'별로 존재해야 하지만, 폭발적인 조합은 막을 수 있다.

스택은 루아에 의해 관리되므로 가비지 콜렉터는 어떤 값을 C가 쓰고 있는지 알 수 있다.

거의 모든 API 함수는 스택을 쓴다.

Pushing elements

타입별로 함수가 있다

lua_pushnil() -> nil

lua_pushnumber() -> doubles

lua_pushinteger() -> integers

lua_pushboolean() -> boolean

lua_pushlstring() -> char* + length

lua_pushstring() -> zero-terminated string

사용자 정의 값을 넣는 함수도 있음.

스택에 원소를 넣을 때마다 충분한 공간이 있는지 확인해야 한다.

처음 시작할 때 스택에는 20개의 빈공간이 준비된다. (LUA_MINSTACK)

보통은 충분한 값이지만, 특히 많은 인자 등을 전달할 때는 다음 함수로 체크하자

```
int lua_checkstack( lua_State* L, int sz )
```

Querying elements

인덱스 규칙

1 - 스택에 제일 처음 추가된 원소

2 - 그 다음에 추가된 원소

-1 - 스택의 꼭대기에 있는 원소 (마지막에 추가된 원소)

-2 - 그 아래 있는 원소

원소의 타입을 체크하는 함수

```
int lua_is* (lua_State *L, int index );
```

해당 타입인지를 검사한다기 보다는, 해당 타입으로 변환될 수 있는지를 확인

```
lua_type()
```

LUA_TNIL, LUA_TBOOLEAN, LUA_TNUMBER 등을 반환한다.

스택에서 값 꺼내오기

```
lua_toboolean()
```

```
lua_tonumber()
```

```
lua_tointeger()
```

```
lua_tolstring() -> 포인터 꺼내주는 게 아니라 복사해주는 것
```

```
lua_objlen()
```

해당 타입으로 꺼내올 수 없을 때는 NULL을 반환.

Other stack operations

스택을 다루는 함수들

lua_gettop() - 스택안의 원소 개수 반환 (꼭대기 원소의 인덱스기도 하다)

lua_settop() - 원소의 개수 설정 (nil을 채우거나, 위쪽의 원소를 삭제)

lua_pushvalue() - 지정한 인덱스의 원소를 복사해서 꼭대기에 푸쉬

lua_remove() - 지정한 인덱스의 원소 삭제, 위쪽의 원소들을 한칸씩 내린다.

lua_insert() - 꼭대기 원소를 지정한 인덱스로 보낸다. 위쪽의 원소들을 한칸씩 올린다.

lua_replace() - 꼭대기 원소를 꺼내서 지정한 인덱스의 원소를 덮어쓴다

24.3 Error Handling with the C API

2007년 6월 14일 목요일

오전 11:05

Error handling in application code

`lua_atpanic()`

루아가 메모리 부족 에러 등을 만날 때 이 API를 사용해서 등록된 함수를 호출한다.

함수를 호출한 후 어플리케이션을 종료한다.

종료를 원하지 않으면 `longjmp` 등을 사용해서 함수가 반환하지 않게 해야 한다.

`lua_pcall()` 을 사용하면 루아 코드가 보호 모드로 실행한다.

에러 발생시 에러 코드를 반환한다.

Error handling in library code

기본적으로 루아는 안전, C 코드를 끼워 넣을 때 문제가 생김.

C 코드를 안전하게 써야 함 `-- poke()` 같은 것을 호출하지 않는 등.

에러를 발견하면 `lua_error()` 혹은 `luaL_error()` 를 호출

에러가 발생해서 `lua_pcall()`이 시작된 부분으로 점프할 때 정리해야 할 것들도 정리해 줌.

25 Extending Your Application

2007년 6월 14일 목요일

오전 11:55

컨피규레이션 언어로서의 루아의 활용

25.1 The Basics

2007년 6월 14일 목요일

오전 11:56

Lua 파일에서 윈도우의 크기를 설정하고, C 에서 읽는 예제를 보자
루아 파일은 이렇게

```
-- define window size
width = 200
height = 300
```

C 코드는 이렇게

```
void load (lua_State* L, const char*fname, int *w, int *h ) {
    if (luaL_loadfile(L, name) || lua_pcall( L, 0, 0, 0 ))
        error(L, "cannot run config. file: %s", lua_tostring(L, -1));
    lua_getglobal(L, "width");
    lua_getglobal(L, "height");
    if ( !lua_isnumber(L, -2))
        error(L, "'width' should be a number\n");
    if ( !lua_isnumber(L, -1))
        error(L, "'height' should be a number\n");
    *w = lua_tointeger(L, -2);
    *h = lua_tointeger(L, -1);
}
```

이렇게 간단한 일을 하는데 루아가 필요할까?

1. 루아가 모든 문법적인 세부사항을 처리한다.

설정 파일이 주석도 가질 수 있다!

C 코드 수정 없이 다양한 확장도 가능하다.

```
-- configuration file
if getenv("DISPLAY") == ":0.0" then
    width = 300; height = 300
else
    width = 200; height = 200
```

2. 새로운 설정을 추가하기가 쉽다. 프로그램이 보다 유연해진다.

25.2 Table Manipulation

2007년 6월 14일 목요일

오후 12:09

윈도우의 배경색을 추가해보자.

```
-- configuration file
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```

이런 방식의 단점.

배경 말고 타이틀, 윈도우의 색상등을 다 설정하려면 너무 길어진다.

background=WHITE 처럼 미리 정의된 색상을 대입할 수도 없다.

테이블을 사용하면 된다.

```
background = {r=0.30, g=0.10, b=0}
```

혹은 이렇게

```
BLUE = { r=0, g=0, b=1}
< other color definitions >
background =BLUE
```

이걸 읽는 C 코드는

```
lua_getglobal(L, "background");
if (!lua_istable(L, -1))
    error(L, "'background' is not a table");
red = getfield(L, "r");
green = getfield(L, "g");
blue = getfield(L, "b");
```

그러나 getfield() 함수는 루아 API에 존재하지 않는다.

우리가 직접 만들어야 하고, 역시나 타입별로 만들어야 한다.

getfield() 함수 코드

```
#define MAX_COLOR 255
/*assume that table is on the stack top*/
int getfield (lua_State *L, const char *key) {
    int result;
    lua_pushstring(L, key);
    lua_gettable(L, -2); /* get background[key] */
    if (!lua_isnumber(L, -1))
        error(L, "invalid component in background color");
    result = (int)lua_tonumber(L, -1) * MAX_COLOR;
    lua_pop(L, 1); /* remove number */
    return result;
}
```

C 어플리케이션에서 미리 일반적인 색상 이름을 정의하는 방법도 있다.

생략.

25.3 Calling Lua Functions

2007년 6월 14일 목요일

오후 1:52

루아 파일에 정의한 함수를 호출해보자.

다음은 루아 함수.

```
function f(x, y)
    return (x^2 * math.sin(y))/(1-x_
end
```

루아 환경이 초기화 되었고, 위의 루아 파일이 로드되었다는 가정하에, 루아 함수 f를 감싸는

C 함수는 이렇게

```
/* call a function 'f' defined in Lua */
double f (double x, double y) {
    double z;

    /* push functions and arguments */
    lua_getglobal(L, "f"); /* function to be called */
    lua_pushnumber(L, x); /* push 1st argument */
    lua_pushnumber(L, y); /* push 2nd argument */

    /* do the call (2 arguments, 1 result ) */
    if (lua_pcall(L, 2, 1, 0) != 0 )
        error(L, "error running function 'f': %s",
              lua_tostring(L, -1));

    /* retrieve result */
    if (!lua_isnumber(L, -1))
        error(L, "function 'f' must return a number");
    z = lua_tonumber(L, -1);
    lua_pop(L, 1); /* pop returned value */
    return z;
}
```

26 Callic C from Lua

2007년 6월 14일 목요일

오후 1:57

루아가 C 함수를 호출할 때도 스택을 사용한다.

이 스택은 글로벌하지 않고, 해당 호출에만 사용되는 private local stack이다.

루아가 C 함수를 호출할 때 첫번째 인자는 무조건 인덱스 1이 된다.

26.1 C Functions

2007년 6월 14일 목요일

오후 2:08

사인값을 구해주는 함수를 예로 만들어보자.

```
static int l_sin (lua_State *L) {
    double d = lua_tonumber(L, 1); /* get argument */
    lua_pushnumber(L, sin(d)); /* push result */
    return 1; /* number of results */
}
```

루아에 등록하기 위해서 다음의 원형을 지켜야 한다.

```
typedef in (*lua_CFunction) (lua_State *L );
```

C 함수를 루아에 등록하자

```
lua_pushcfunction(L, l_sin);
lua_setglobal(L, "mysin");
```

보다 전문적인 사인 함수로 만들어보자. 인자를 체크.

```
static int l_sin (lua_State *L) {
    double d = lua_checknumber(L, 1);
    lua_pushnumber(L, sin(d));
    return 1; /* number of results */
}
```

만약 `mysin('a')` 처럼 호출했다면 다음의 에러를 발생시킨다.

```
bad argument #1 to 'mysin' (number expected, got string)
```

26.1 C Modules

2007년 6월 14일 목요일

오후 2:13

루아 모듈은 루아 함수들을 정의하고, 테이블에 저장해놓은 청크에 불과하다.
C 함수를 사용해서 루아를 확장할 때, 모듈로서 제공하는 것은 좋은 아이디어.
luaL_register() 함수는 함수와 함수이름의 배열을 루아에 등록시켜준다.

```
static const struct luaL_Reg mylib[] = {  
    {"mysin", l_sin},  
    {NULL, NULL} /*sentinel */
```

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

luaL_register() 는 주어진 이름("mylib")로 테이블을 만들고, 이름-함수 페어로 채운다.

이 코드를 dll(혹은 so)로 만들어 적절한 경로에 등록하고, 루아에서 다음과 같이

```
require "mylib"
```

적절한 dll을 찾고, luaopen_mylib() 함수를 꺼내서 호출한다.

27 Techniques for Writing C Functions

2007년 6월 14일 목요일

오후 2:21

생략

28 User-Defined Types in C

2007년 6월 14일 목요일

오후 2:22

생략

29 Managing Resources

2007년 6월 14일 목요일

오후 2:23

30 Threads and States

2007년 6월 14일 목요일

오후 2:23

생략

31 Memory Management

2007년 6월 14일 목요일

오후 2:24

생략

메모리 할당 함수를 C에서 제공.