Game Programming with Lua

afewhee@gmail.com

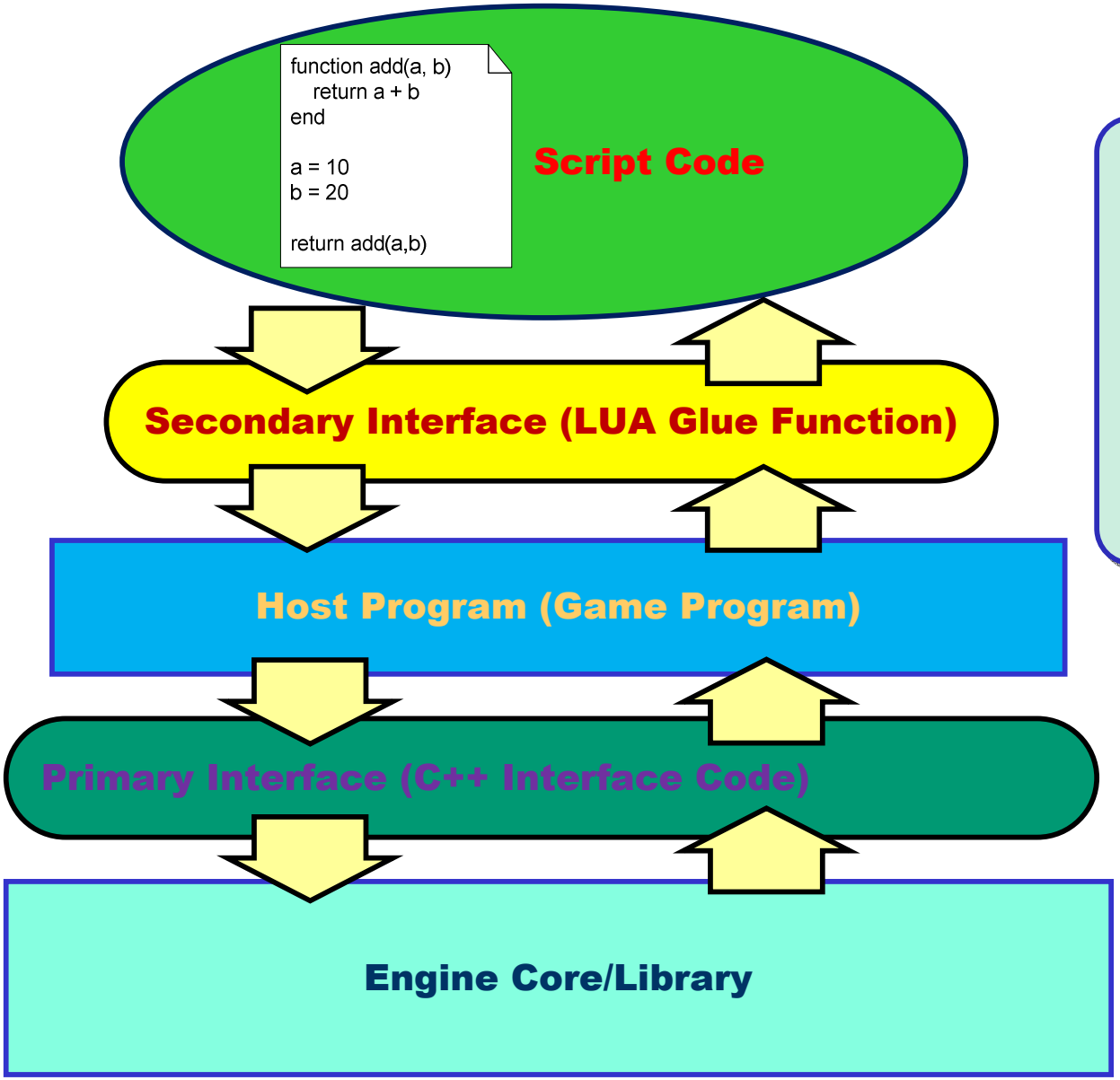
0 스크립트 언어와 Lua

- 스크립트 언어란?
  - 인터프리터에 의해 번역
  - 컴파일 되지 않은 언어 ➔ 번역 즉시 실행
  - 인터프리터 필요(Interpreter) ➔ 언어가 독립적으로 실행 되지 않고 인터프리터가 내장된 프로그램에 의해 실행

- 스크립트 언어의 목적
  - 개발 기간 및 개발 비용 단축, 비 숙련성
  - 자주 갱신해야 하는 복잡한 프로그램

- 스크립트 언어들
  - 웹: 펄, PHP, ASP, JSP, VBA, 자바스크립트
  - 게임: 루아, 파이썬

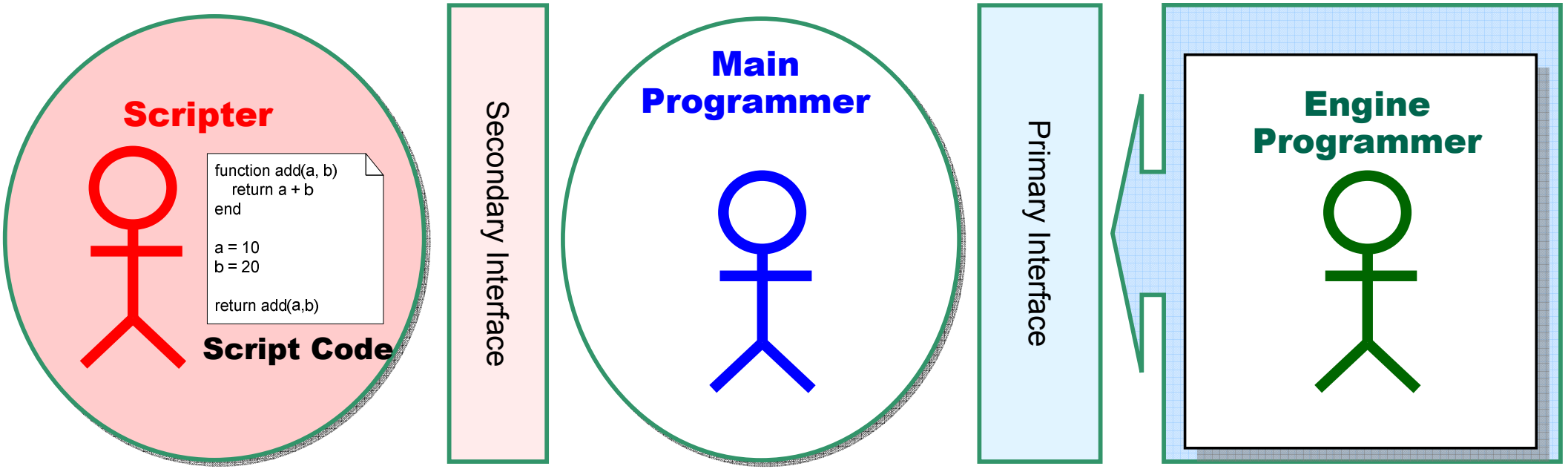
게임 시스템에서 Lua의 위치



Script 코드는 응용 프로그램과 인터페이스를 통해서 통신

응용 프로그램은 스크립트 해석에 대한 인터프리터, 실행에 대한 코드 포함

● 게임 개발자 관계



게임에서의 레벨 작업, UI, 데이터 등을 스크립트로 작성

응용 프로그램인 게임 프로그램을 작성하고 스크립트를 위해 Interface와 실행 코드 제작

게임 프로그램에 필요한 필수 라이브러리, Utility등 장르와 실행 플랫폼에 덜 의존적인 독립성이 보장되는 코드 작성 응용 프로그램을 위한 인터페이스 제공

- 확장성:
  - 루아 코드 모두를 통해 외부의 C 코드를 통해 확장할 수로 처음부터 설계.
  - 루아 API용 함수 작성이 용이.
  - C / C + +과 다른 스크립트 언어, 포트란, 자바, 스몰 토크, ADA(에이다)와 같은 다른 언어로도 확장 가능
- 단순성:
  - 문법이 간단.
  - 스크립트 실행기의 크기가 작음(1MB)이하. --> 소형 임베디드 환경에서도 훌륭하게 동작
- 이식성:
  - ANSI C로 작성되어 있어서 유닉스, 리눅스, 윈도우 등 워크스테이션 뿐만 아니라 임베디드 환경에도 코드의 이식성 높음 --> 사용의 대역대가 넓음
- 라이센스:
  - Open Source. 라이센스 비용 없이 상업용, 비 상업용 용도로 수정해서 사용가능

1 언어 (Language)

- Lua home page: http://lua.org

- lua.exe
  - 루아 내장 함수로 간단한 문장 출력
    - *print("Hello World")*
  - 다소 긴 문장도 실행

```
-- 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))
```

  - 스크립트 파일 실행
    - *prompt> lua hello.lua*

- 청크: 루아에서 실행되는 명령 단위: line, 함수, 파일

- line:

```
a = 1
b = a*2

a = 1;
b = a*2;

a = 1 ; b = a*2     -- 한 줄에 2개의 명령 실행 semicolon 사용
a = 1   b = a*2     -- 허용 한됨
```

- 함수:

```
print("Hello world")
```

- 파일: 스크립트 파일 로드 --> dofile() 사용

```
dofile("lib1.lua")   -- load your library
n = norm(3.4, 1.0)
print(twice(n))      --> 7.0880180586677
```


- 식별자(identifier): 사용자가 프로그램을 위해 사용하는 단어.
- 변수, 함수, 테이블, 클래스, 메서드, 구조체 등의 이름 등
- 루아 식별자: 숫자로 시작하지 않는 영문자, 숫자, 밑줄('_')의 조합으로 사용

  i j i10 _ij
  aSomewhatLongName _INPUT
- 예약어(Keyword): 컴파일러 또는 인터프리터에서 미리 지정된 단어.
- 예약어를 식별자로 사용 불가.

- 루아 예약어:

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

연산자: + - * / %
주석:
한줄 --
여러줄 '--[[' ']]' 또는 '--[[' '--]]'
기타: {}

- 루아는 대소 문자 구분함:
  - and는 예약어 이지만 AND는 예약어가 아니므로 식별자로 사용가능

- 주석 요령

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

---[[            -- 하이픈('-') 이 추가되어 이 줄만 주석 처리됨
print(10)        --> 실행 됨
--]]
```


- 전역 변수는 타입 선언을 하지 않음.
- 루아는 기본이 전역 변수.

- 초기화 되지 않은 변수는 오류가 아님 대신 nil 반환

print(b) --> 초기화 안된 b는 nil
b = 10
print(b) --> b는 이전에 10으로 설정되어 이 값을 출력

- 전역 변수 제거 작업은 필요 없음.
  - 꼭 전역 변수를 지워야 한다면 nil을 배정

- 속도를 위해서 변수의 생명 주기를 줄이고자 한다면 지역 변수 예약어 'local'을 사용

```
a = 10              --> 전역 변수

function MyFunction()
local b = 20        --> 지역 변수
...
end
```

- 사용법 : lua [options] [script [args]]
- 옵션 보기: lua --help
- -e 옵션: 명령 라인에 직접 코드를 입력.
  prompt> lua -e "print(math.sin(12))" --> -0.53657291800043
- -l 옵션: 라이브러리 로드
  prompt> lua -i -l lib1.lua -e "x = 10"
- 배정 문자('='): 특정 구문의 결과 값 출력

  > = math.sin(3) --> 0.14112000805987
  > a = 30
  > = a --> a 30

- lua.exe는 스크립트 시작 전 arg 테이블을 생성하고 모든 명령행 인수를 저장

prompt> lua -e "sin=math.sin" script a b

- arg배열에는 다음과 같이 저장.

arg[-3] = "lua"
arg[-2] = "-e"
arg[-1] = "sin=math.sin"
arg[0] = "script"
arg[1] = "a"
arg[2] = "b"

2 타입과 값 (Types and Values)

- 바인딩: 변수 또는 함수에 속성을 부여하는 것

- 속성: 이름(Type), 형(Type), 값(Value), 범위(Scope), 생명 시간(life time), 저장 위치(storage class)

- 동적 바인딩: 프로그램 실행 중에 속성을 정하는 것
  - 대부분의 스크립트 언어들은 동적 바인딩을 사용 ➔
  - 동적 타입 지정 언어(dynamic typed language): 프로그램 실행 중에 속성을 결정. ➔사용 전에는 타입이 결정되어 있지 않음.

- 루아의 기본 데이터 타입 8종류:
  - 닐(nil), 부울형(boolean), 수치(number),
  - 문자열(string), 사용자 데이터(userdata),
  - 함수형(function), 스레드(thread), 테이블(table)

```
--타입 출력 예1)
print(type("Hello world"))   --> string
print(type(10.4*3))          --> number
print(type(print))           --> function
print(type(type))            --> function
print(type(true))            --> boolean
print(type(nil))             --> nil
print(type(type(X)))         --> string: 'type'의 타입은 문자열임
```

- 다른 타입과 다르다는 것을 나타내는 목적으로 사용
- nil은 단 하나의 값인 'nil' 을 지정하는 타입
- 전역에 nil 값을 적용하면 지워짐
- 테이블 객체 등을 지울 때도 'nil' 을 사용
- 초기화 안된 값, 유용한 값이 아니거나 정상 값이 아닐 때 사용

- 조건 값 false(거짓), true(참) 만 가지는 타입
- 조건 검사에서 false, nil 두 값만 거짓으로 판별됨
- 주의!!) 숫자 0, 빈 문자열("")은 참으로 판정
  - C 언어는 0은 false로 판정함으로 숫자 0과 혼용해서 사용하기 보다는 숫자면 숫자, boolean이면 boolean 으로 만 판정하는 것이 코드 작성에 유리

- 배정도 소수점(double-precision floating point: 64bit)을 나타냄

- 32비트 크기의 숫자에는 반올림 오차가 발생 없음.
  - -2^31 ~ 2^31-1 까지 정수형으로 사용할 수 있음.

- 정수, 실수, 지수 표현 모두 가능

수치 표현 예)

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

- 문자열
  - 문자 배열을 의미. 큰 따옴표 "" 또는 작은 따옴표 '' 이용

- 루아의 문자열은 변경 불가

- 변경하려면 string 객체 함수 사용해서 교체 후 다른 문자열에 복사

```
a = "one string"
b = string.gsub(a, "one", "another")  -- gsub: a 문자열 안에 있는 "one"을 "another"로 변경 한 후 b에 복사
print(a)        --> one string
print(b)        --> another string
```

- **escape 문자**

\a bell
\b back space
\f form feed
\n newline
\r carriage return
\t horizontal tab
\v vertical tab
\\ backslash
\" double quote
\' single quote
\[left square bracket
\] right square bracket
\ddd 10진수 지정(항상 3자리씩 읽음: 주의)

```
> print("one line\nnext line\n\"in quotes\", 'in quotes'")
one line
next line
"in quotes", 'in quotes'

> print('a backslash inside quotes: \'\\\'')
a backslash inside quotes: '\'

> print("a simpler way: '\\'")
a simpler way: '\'
```


- 수치의 자동 변환

```
print("10" + 1)          --> 11
print("10 + 1")          --> 10 + 1
print("-5.3e-10"*"2")    --> -1.06e-09
print("hello" + 1)       -- ERROR (cannot convert "hello")
```

- 문자 병합:
  - '..' 전후에는 반드시 공백 필요

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

- 강제로 형을 변환시키면 프로그램이 복잡해질 수 있음.

- tonumber(): 문자를 수치로 변경

12345678 == "12345678"의 비교 결과는 false

```
>a = "12345678"
>print(type(a))        --> string

>a = tonumber(a)
>print(type(a))        -->number
>print(a)              -->12345678
```

- tostring(): 수치를 문자열로 변경

```
print(tostring(10) == "10")      --> true
print(10 .. "" == "10")          --> true
```

- 문자열의 길이: string.len()
  - 5.1이후 # 이용

```
>a = "Hello world"   -- a에 "Hello world" 대입
>a = #a              -- a 문자열 길이를 a에 다시 저장. a는 수치로 바뀜
>print(a)            -->11
```


- 연관 배열(asociative array:
  - {키, 값}한 쌍으로 구성)를 구현한 객체(Object).
  - 연관이란 키(key)를 가지고 자료를 순회하는 구조
- 루아의 유일한 자료구조 기능
  - 테이블을 사용해서 배열, 심벌 테이블, 집합(set), 레코드(record), 큐(queue) 등의 자료구조가 가능함으로 모듈, 패키지, 객체등을 표현할 수 있음.
- 테이블 객체 생성: {} 사용
- 테이블 객체는 소유권이 정해지지 않음
- table 참조가 더 이상 없으면 자동으로 garbage collector에 의해 테이블이 해제됨
- table은 새로운 키가 생성될 때 마다 증가

- 예)

```
a = {}   -- 테이블 객체를 생성하고 a 에 테이블 참조

k = "x"

a[ k] = 10            -- 키가 "x"이고 이 키에 값 10을 바인딩
a[20] = "great"       -- 키가 20 이고 이 키에 "great"라는 문자열을 바인딩

print(a["x"]     )    --> a가 참조하는 테이블에 키가 "x"로 바인딩된 값이 출력됨

k = 20                -- k에 수치 20 배정
print(a[k])           --> a가 참조하는 테이블에 키가 20으로 바인딩된 값 출력

a["x"] = a["x"] + 1 -- a가 참조하는 테이블의 키가 "x"으로 바인딩된 값을 1 증가 시킴
print(a["x"])         --> 11
```

- 테이블 객체는 소유권이 정해지지 않음

```
a = {}                -- 테이블 객체를 생성하고 a로 참조
a["x"] = 10           -- key "x"에 수치 값 10 바인딩

b = a                 -- b는 a 가 참조하는 table 객체 참조
a = nil               -- a는 더이상 table을 참조하지 않음. b는 여전히 참조
a = nil               -- b도 더이상 table을 참조하지 않음.
```


- table은 새로운 키가 생성될 때 마다 증가

```
a = {}      -- empty table

-- 새로운 항목 100개 생성
for i=1,1000 do a[i] = i*2 end

print(a[9])      --> 18
a["x"] = 10      -- 키 "x"에 10 할당.
print(a["x"])    --> 10
print(a["y"])    --> nil  key "y"에 값이 바인딩되지 않아 nil 출력
```

- .연산자와 [] 연산자 관계

```
a.x = 10          -- a["x"] = 10 과 동일
a.y = 20          -- a["y"] = 20 과 동일
print(a.x)        -- print(a["x"]) 와 동일
print(a.y)        -- print(a["y"]) 와 동일
```


- 테이블 사용시 주의점
  - 1. 루아의 배열 인덱스는 1부터 시작 # 사용 시 주의
  - 2. 길이 '#' 사용 주의

```
a={}
a[0] =1 --> 문법 에러는 아니지만 #a하면 0으로 출력됨

a={}
a[1000] = 1도 #a 값은 0으로 출력됨을 주의!
이런경우 table.maxn()로 확인.

table.maxn(a)    --> 1000
```

  - 3. 키 사용시 주의

a.x와 a[x]는 같지 않음. --> a.x 는 a["x"]을 의미
a["+0"], a["-0"], a["0"] --> 각각의 키가 "+0", "-0", "0"을 의미하기 때문에 다름

- 함수
  - 루아 함수는 일등급 값: --> 루아 함수는 변수, 테이블, 인수, 또는 다른 함수의 반환 값 등으로 사용할 수 있음 ➔6장 참조
  - 글루 함수:
    - lua 확장 함수로 lua에서 사용할 수 있도록 C언어 등으로 만든 함수
  - lua API:
    - C언어 등에서 사용하는 Lua 스크립트로 작성된 함수.
- 사용자 정의 데이터
  - 루아는 C언어로 만든 사용자 정의 데이터를 사용할 수 있음 --> 예) 게임 데이터, 파일 입/출력 데이터
- 스레드
  - 9장 코루틴 참조

3 수식 (Expressions)

3.1 - 산술 연산자

- 덧셈: '+' 뺄셈 '-', 곱셈 '*', 나눗셈 '/' 거듭제곱 '^' 나머지 '%'

- 거듭제곱: 지수 승
  - 제곱근: x^0.5
  - 세제곱근: x^(1/3)

- 나머지 연산자: 정수와 실수일 때 다르게 사용
  - 정수: 정수형 나머지. c언어의 %(modular)연산자와 동일
    - *a % b = a = floor(a/b) * b*
  - 실수: x%0.01은 x의 소수점 3자리 이후부터의 나머지 값

```
x = math.pi
print(x - x%0.01)      -->3.14


-- 임의의 각도를 [0, 2pi)로 정규화 하기
local tolerance = 0.17
function isturnback(angle)
  angle = angle %(2 * math.pi)
  return (math.abs(angle - math.pi) < tolerance
end
```

주의) lua math의 삼각함수는 degree 대신 radian 를 사용

- 루아 관계 연산자: < > <= >= == ~=
- 모든 관계 연산자는 true 또는 false 반환

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

print(a == c)        --> true
print(a == b)        --> false
```

- 비교를 할 때 타입에서 주의
  - 2 < 15는 true 이지만 "2" < "15" false 임
  - 2 < "15" 는 수치, 문자열로 다른 타입을 비교함으로 프로그램 오류

- 제어 구조: 'and', 'or', 'not'
  - false, nil 는 거짓, 나머지 값은 모두 참(수치 값 0도 참임)

- and: false 일 경우 첫 번째 인자 반환. 그렇지 않으면 두 번째 인자 반환
- or : false 가 아닐 경우 첫 번째 인자 반환. 그렇지 않으면 두 번째 인자 반환
- not: 언제나 true 또는 false 반환

```
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

print(not nil)          --> true
print(not false)        --> true
print(not 0)            --> false
print(not not nil)      --> false
```


- and, or 모두 단축(short_cut) 계산 방식.
  - 필요할 때만 두 번째 인자를 계산
  - and) a == b and b == c 에서 a==b 가 거짓이면 b == c는 계산 안함
  - or) a == b or b == c 에서 a==b 가 참이면 b == c는 계산 안함

--존재하는 값으로 대치

x= x or v --> if not x then x = v end

--두 수의 최대값 구하기

max = (x > y) and x or y

※ x > y이면(참이면), x가 수치이면 x도 참이므로 and는 2번째인 x반환
x<y이면 (x > y) and x 가 거짓이므로 y값을 반환

- 수치, 문자열 결합 결과는 문자열

```
print("Hello " .. "World")   --> Hello World
print(0 .. 1)                --> 01
```

- 루아는 문자열을 변경 할 수 없으므로 ..의 결과 값은 항상 새로운 문자열

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

- ^
- not - (unary)
- * / %
- + -
- ..
- < > <= >= ~= ==
- and
- or

- '^', '..': 오른쪽이 우선
- 나머지 연산자: 왼쪽이 우선

```
a+i < b/2+1          -->   (a+i) < ((b/2)+1)
5+x^2*8              -->   5+((x^2)*8)
a < y and y <= z     -->   (a < y) and (y <= z)
-x^2                 -->   -(x^2)
x^y^z                -->   x^(y^z)
```

- 테이블 생성자: 생성과 동시에 초기화

예1)

days = {"Sunday", "Monday", "Tuesday"} --> days = {}; days[1] = "Sunday"; days[2] = "Monday"; days[3] = "Tuesday"}

예2)

a = { x= 10, y = 20} -->a = {}; a.x=0; a.y=0

- 테이블을 만들 때, 어떤 종류의 생성자를 사용하더라도, 언제든지 생성된 테이블에서 필드를 추가하거나 제거 가능

```
w = {x=0, y=0, label="console"}
x = {sin(0), sin(1), sin(2)}
w[1] = "another field"
x.f = w
print(w["x"])    --> 0
print(w[1])      --> another field
print(x.f[1])    --> another field
w.x = nil        --> remove field "x"
```


Linked List 예)

```
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["color"])   --> blue
print(polyline[2].x)         --> -10
print(polyline[4].y)         --> 1
```

- 각 괄호 안에 초기화할 키를 수식으로 직접 넣는 방법

```
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])          --> ---
```

- 배열의 인덱스 0에 값을 할당할 수 있으나 이 값은 다른 필드에 영향이 없음. 길이 검색에서 제외됨.

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

print(days[0])  --> Sunday
print(#days)  --> 6
```


- 테이블 마지막에 쉼표(,)를 넣을 수 있음

a = {[1]="red", [2]="green", [3]="blue",}

- 쉼표 대신 세미 콜론도 사용가능
  - '#'으로 길이 값 을 구하면 ';' 이후부터의 리스트 숫자 반환

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

print(a.x) -->10
print(a.y) -->45
print(#a) -->3

4 문장 (Statements)

- 루아는 파스칼(Pascal)과 비슷한 형태인 문장 구조
- 다중 값 할당 가능

-- 일반적인 값 할당

```
a = "hello" .. "world"
t.n = t.n + 1
```

- 다중 값 할당: 쉼표(‘,‘) 사용

```
a, b = 10, 2*x  <==> a =10; b = 2*x
```

-- 다중 값 할당을 이용한 값 교환(swap)

```
x, y = y, x                    -- x , y 교환
a[i], a[j] = a[j], a[i]        -- a[i], a[j] 교환
```


- 다중 값 할당 시 할당 받는 변수의 개수가 많으면 nil로 채움

```
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
```

- 루아 함수는 2개 이상 반환 가능 --> 이 때 다중 값 할당 사용

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


- 지역 변수: 자신을 선언한 chunk 안에서 유효한 변수
- local 키워드 사용
- 지역 변수 사용은 속도에 이득
- 초기화 안하면 nil 값이 배정

```
x = 10              -- 전역 변수
local i = 1         -- 지역 변수

  while i<=x do
    local x = i*2  -- while 안에서 유효한 지역 변수
    print(x)     --> 2, 4, 6, 8, ...
    i = i + 1
  end

if i > 20 then
  local x          -- "then" 본체에서 유효한 지역 변수
  x = 20
  print(x + 2)
else
  print(x)       --> 10  (the global one)
end

print(x)         --> 10  (the global one)
```

- do-end 블록을 사용한 지역 변수

```
do
  local a2 = 2*a
  local d = sqrt(b^2 - 4*a*c)
  x1 = (-b + d)/a2
  x2 = (-b - d)/a2
end                -- a2, d의 유효 범위 끝

print(x1, x2)
```

- 같은 이름으로 전역 값을 지역에 복사하고 사용

```
f = 10
do
  local f = f
  print(f)      -- 10
  f = f+20
  print(f)      -- 30
end

print(f)        -- 10
```


- 키워드: if, while, repeat, for, until, end, break, return
  - 루아는 nil, false만 거짓. 나머지 전부 참.
  - 수치 0, 빈 문자열 ""도 참임

- 4.3.1 if '조건' then elseif '조건' then else end

```
if a<0 then a = 0 end

if a<b then return a else return b end

if line > MAXLINES then
  showpage()
  line = 0
end
```

```
if op == "+" then
   r = a + b
elseif op == "-" then
   r = a - b
elseif op == "*" then
   r = a*b
elseif op == "/" then
   r = a/b
else
   error("invalid operation")
end
```

```
if op == "+" then
   r = a + b
elseif op == "-" then
   r = a - b
elseif op == "*" then
   r = a*b
elseif op == "/" then
   r = a/b
else
   error("invalid operation")
end
```

- while '조건' end

- 조건이 거짓이면 반복 종료

```
local i = 1
while a[i] do
  print(a[i])

  i = i + 1
end
```

- repeat until '조건'
  - while과 비슷하나 repeat 블록 코드는 처음 1번은 무조건 실행
  - repeat 블록 내의 local 변수는 until 조건문까지 유효

```
local sqr  =x/2
repeat
   sqr = (sqr + x/sqr)/2
   local err = math.abs(sqr -x)
until err < x/1000              -- err는 여기에서도 유효
```

- Numeric (수치) for 문
  - Statements

```
for var=시작 값, 끝 값, 증가 값 do
  do-something
end

-- find a value in a list
local found = nil
for i=1,a.n do
  if a[i] == value then
    found = i       -- save value of `i'
    break
  end
end

print(found)
```

  - 증가 값이 생략되면 자동으로 1씩 증가
  - 주의) for 문 제어용으로 사용되는 변수는 지역 변수임

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

print(i)     -- nil 출력
```

- Generic (일반) for 문
  - 반복자(iterator) 함수에서 반환된 모든 값을 순회
  - 반복자 함수 ipairs()는 테이블의 색인(키)에 대한 값 반환
  - for-each와 유사
  - 7.2장 참조

- Statements

```
for <변수 목록> in <수식 목록> do
  do-something
end

-- 색인에 대한 값 출력
days = {"Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"}

for i,v in ipairs(days) do print(i, v) end

-- 색인 출력
for v in ipairs(days) do print(v) end
```


- Generic (일반) for 문을 사용한 역 참조

```
-- 역 참조(키와 값의 쌍을 값-키값으로 변경) 테이블 만들기
revDays = {["Sunday"] = 1, ["Monday"] = 2,
           ["Tuesday"] = 3, ["Wednesday"] = 4,
           ["Thursday"] = 5, ["Friday"] = 6,
           ["Saturday"] = 7}

x = "Tuesday"
print(revDays[x])     --> 3


-- 간단히 generic for로 해결

revDays = {}
for i,v in ipairs(days) do
  revDays[v] = i
end
```


- break: 반복문을 빠져 나갈 때
- return: 단순히 함수를 빠져 나갈 때 또는 값을 반환하고 함수를 종료 할 때
- return문은 조건문, do-end 블록, 함수 끝에서만 사용

```
local i = 1
while a[i] do
  if a[i] == v then break end     -- while을 종료
  i = i + 1
end

function foo ()

  return                  -- 문법 오류
  ...

  if x== 10 then
    return                -- OK
  end

  do return end           -- OK

end
```

5 함수 개요 (Function)

- 함수: 작업의 실행 단위
- 인수 전달: 소괄호를 사용
- 인수가 하나이고 문자열, 또는 테이블 생성자이면 '()'를 생략

```
print(8*9, 9/8)                         -- 인수를 전달한 함수 호출
a = math.sin(3) + math.cos(10)          -- 수식과 같이 사용되는 함수 호출
print(os.date())                        -- 함수의 결과를 인수로 함수 호출

()를 생략한 함수 호출 방법
print "Hello World"              --    print("Hello World")
dofile 'a.lua'                  --    dofile ('a.lua')
print [[a multi-line            --    print([[a multi-line
      message]]                               message]])
f{x=10, y=20}                   --    f({x=10, y=20})
type{}                          --    type({})
```


```
function '함수 이름'(인수 리스트)
   do-somthing
end
```

```
-- a 배열의 모든 값 더하기
function sum (a)
   local s = 0
   for i,v in ipairs(a) do
   s = s + v
   end
   return s
end
```

- C언어는 실 인수, 형식 인수의 개수가 같아야 하지만 루아 함수는 인수의 숫자가 같이 않아도 함수 함수 호출 가능

- 실 인수 개수가 형식 인수 개수를 초과하면 초과된 인수는 버림

```
function MyFunc(a, b)
  return  a or b or 1
end
```

호출	인수	결과
MyFunc()	a=nil, b=nil	--> 1
MyFunc(3)	a=3 , b=nil	--> 3
MyFunc(nil, 4)	a=nil, b=4	--> 4
MyFunc(3, 4)	a=3 , b=4	--> 3
MyFunc(3, 4, 5)	a=3 , b=4 (5는 버림)	--> 3

- ruturn 키워드 다음에 반환 값 나열
- return 값1, 값2, . . .

ex)

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

```
function maximum (a)
  local mi = 1             -- maximum index
  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
```

```
-- 다음과 같이 함수가 정의되어 있을 때
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'는 버림
x,y,z = 10,foo2()      -- x=10, y='a', z='b'

-- 함수의 결과 값이 없거나 필요한 만큼 반환 받지 못하면 nil로 채움
x,y    = foo0()        -- x=nil, y=nil
x,y    = foo1()        -- x='a', y=nil
x,y,z = foo2()         -- x='a', y='b', z=nil

-- 목록에서 마지막 요소가 아닌 함수 호출은 언제나 결과값 하나만 반환
x,y = foo2(), 20       -- x='a', y=20
x,y = foo0(), 20, 30   -- x='nil', y=20, 30 버림

-- 함수가 다른 함수의 인수로 호출될 때 마지막 인수가 되면 모든 반환 값이 후자의 인수로 전달
print(foo0())          -->
print(foo1())          --> a
print(foo2())          --> a   b (함수 호출이 마지막이므로 전부 반환된 값이 인수로 전달)
print(foo2(), 1)       --> a   1 (함수 호출이 마지막이 아니므로 하나만 반환
print(foo2() .. "x")   --> ax
```


```
-- 여러 개를 받는 경우 테이블 이용
t = {foo0()}              -- t = {}   (an empty table)
t = {foo1()}              -- t = {'a'}
t = {foo2()}              -- t = {'a', 'b'}

-- 다음과 같은 경우도 목록에서 마지막 요소가 아니므로 함수 호출은 하나만 반환
t = {foo0(), foo2(), 4}    -- t[1] = nil, t[2] = 'a', t[3] = 4

-- 함수에서 반환으로 함수 호출하면 전부 반환
function foo (i)
  if i == 0 then return foo0()
  elseif i == 1 then return foo1()
  elseif i == 2 then return foo2()
  end
end

print(foo(1))      --> a
print(foo(2))      --> a   b
print(foo(0))      -- (no results)
print(foo(3))      -- (no results)

-- 강제로 결과를 1개만 반환하려면 소괄호로 감싼다. 따라서 return 문에 소괄호를 감싸지 않는다.

print((foo0()))          --> nil
print((foo1()))          --> a
print((foo2()))          --> a
```

※return (f(x))와 return f(x)는 차이가 큼

```
-- unpack: 배열을 인수로 받아서 배열의 색인1부터 시작하는 모든 요소를 결과로 반환

print(unpack{10,20,30})      --> 10    20    30
a,b = unpack{10,20,30}       -- a=10,  b=20,  30은 버림

변경 가능한 함수 f를 배열 a에 들어 있는 모든 가능한 값들로 호출할 때
f(unpack(a))

f = string.find
a = {"hello world",  "ld"}
b = f(unpack(a))
print(b)              --> 10

-- 재귀 기법으로 unpack() 구현
function unpack (t,  i)
i = i or 1
if t[i]  ~= nil then
return t[i],  unpack(t,  i + 1)
end
end
```


- 가변 인수 지정 "..."

```
function add(...)
  local s = 0
  for i, v in pairs{...} do
    s = s + v
  end

  return s
end

print(add(3,4,10,25,12))     -->54

-- 가변 인수로 구현한 print
function MyPrint(...)
  local Result = ""
  for i,v in ipairs(arg) do
  Result = Result .. tostring(v) .. "₩t"
  end
  Result = Result .. "₩n"

  print(Result)
end

MyPrint("Hello", "world", 2002, "World", " cup")
```


```
-- select 함수를 사용한 var-arg 나열
function foo(a,b, ...)
  local arg = {..}; arg.n = select("#", ...)
  <함수 본체>
end

--> 5.1 이후 '#' 이용한 방법
function foo(a,b, ...)
  local arg = select("#", ...)
  <함수 본체>
end
```

- Named Arguments: 매개 변수를 인수가 순서가 아닌 인수 이름에 따라 지정하는 방법
- 루아는 이름 있는 인수 문법이 없지만 테이블로 인수를 묶어서 전달하면 Named Arguments 효과를 만들 수 있음

```
-- sudo-code(루아 에서는 실행 안됨)
rename(old="temp.lua", new="temp1.lua")

function rename (arg)                          -- 이름을 바꾸는 함수
  return os.rename(arg.old, arg.new)
end

ren= {old="temp.lua", new="temp1.lua"}     -- 테이블로 인수를 묶는다.
rename(ren)

-- 윈도우 생성 예)
function Window (options)
  -- check mandatory options
     ...
  -- everything else is optional
  CreateWindow(options.title,
       options.x or 0,                       -- default value
       options.y or 0,                       -- default value
       options.width, options.height,
     )
End
w = Window{ x=0, y=0, width=300, height=200, title = "Lua", background="blue", border = true  }
Window(w)
```

6 함수 고급 활용

- 루아의 함수는 일등급 값(First Class Value) --> 함수를 변수처럼 사용

```
a = {p = print}                  -- 함수를 테이블 객체에 저장
a.p("Hello World")  --> Hello World

print = math.sin      -- print는 math.sin 함수를 참조
a.p(print(1))         --> 0.841470

sin = a.p             -- sin 함수는 a.p인 print 함수 참조
sin(10, 20)           --> 10        20
```

- 테이블과 마찬가지로 함수를 구현하고 이를 변수에 저장 가능

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


- Anonymous Function(익명 함수):
  - 함수 이름 없이 function(인수 리스트) 본체 end로 내포(Nasted)된 함수
  - 결과 함수로 저장

```
function MyFunc()
  local a = 10; local b = 20

  return function(a, b) -- 익명 함수
    return a + b
  end
end

c = MyFunc()                -- 변수에 함수 저장
print(c, c(10, 20)))        --> function, 30

--네트워크 ip 리스트에 대한 table 객체의 정렬
network = {
    {name = "grauna",   IP = "210.26.30.34"},
    {name = "arraial",    IP = "210.26.30.23"},
    {name = "lua",       IP = "210.26.23.12"},
    {name = "derain",   IP = "210.26.23.20"},
 }

table.sort(network, function (a,b)
                      return (a.name > b.name)
                    end      )
```


```
-- 미분 함수 정의
function derivative(f, delta)
  delta = delta or 1e-4

  return function(x)
    return ( f(x + delta) -f(x))/delta
  end

end

-- 미분 함수 호출
c = derivative(math.sin)  -- 미분 함수의 f를 sin 함수로 지정
                          -- 변수 c에 저장
print(c)                  --> function 출력
print(c(10))              --> -0.8390443 미분 값 출력
```


```
-- 미분 함수 정의
function derivative(f, delta)
  delta = delta or 1e-4

  return function(x)
    return ( f(x + delta) -f(x))/delta
  end

end

-- 미분 함수 호출
c = derivative(math.sin)  -- 미분 함수의 f를 sin 함수로 지정
                          -- 변수 c에 저장
print(c)                  --> function 출력
print(c(10))              --> -0.8390443 미분 값 출력
```


- 루아의 정적 범위 지정(Lexical scoping): 함수 안에서 함수를 구현하는 경우 내포된 함수는 내포한 함수의 지역변수를 접근할 수 있음.
- 클로저: 하나의 함수와 그 함수가 정확하게 접근해야 하는 모든 비 지역 변수를 합한 것
→ 자신의 범위밖에 있는 변수를 접근할 수 있음.
- 루아의 함수는 클로저. 함수형 프로그램, callback 함수로 사용하기 용이

```
function newCounter ()
  local i = 0
  return function ()   -- 익명 함수
         i = i + 1      -- 익명 함수가 자신의 범위 밖에 있는 변수 i를 접근
         return i
       end
end

c1 = newCounter()
print(c1())  --> 1
print(c1())  --> 2     -- i가 local 이지만 static처럼 계속 유효

c2 = newCounter()
print(c2())  --> 1     -- static과 다른 부분이 해당 클로저에만 유효
print(c1())  --> 3
print(c2())  --> 2
```


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

-- 보안을 위한 io.open 재정의
do
  local oldOpen = io.open
  io.open = function (filename, mode)
  if access_OK(filename, mode) then
    return oldOpen(filename, mode)
  else
    return nil, "access denied"
  end
  end
end
```


- 지역 함수: 함수를 지역 변수에 저장 또는 local 예약어 사용

```
local function()
...
end
```

- 주의 지역 함수는 재귀 호출에서 문제가 될 수 있음

```
local f = function(n)
  return n * function(n-1)  -- buggy
end

-- fact(n-1)을 컴파일 하는 시점에서 지역함수 f는 정의되지 않았으므로 error
-- 다음과 같이 변경

local f
f = function(n)
  return n * function(n-1)
end
```


- Tail call: 함수 호출로 가장한 goto
- 꼬리 호출할 때 추가 스택을 사용하지 않음. --> 꼬리 호출 제거(tail-call elimination)
- 형식: function f(x) return g(x) end

```
function fact (n)
  if n > 0 then
    return fact(n - 1)                    -- 꼬리 호출
  end
end

return x[i].foo(x[j] + a*b, i + j)     -- return function() 형태이므로 Proper Tail Call
```

- 다음은 꼬리 호출이 아님

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

- 다음 모두도 꼬리 호출이 아님

```
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
```


```
--꼬리 호출을 사용한 미로 게임
function room1 ()
  local move = io.read()
  if move == "south" then return room3()
  elseif move == "east" then return room2()
  else print("invalid move")
      return room1()    -- stay in the same room
  end
end

function room2 ()
  local move = io.read()
  if move == "south" then return room4()
  elseif move == "west" then return room1()
  else print("invalid move")
      return room2()
  end
end

function room3 ()
  local move = io.read()
  if move == "north" then return room1()
  elseif move == "east" then return room4()
  else print("invalid move")
      return room3()
  end
end

function room4 ()
  print("congratulations!")
end
```

7 Iterator, Generic for

- 반복자: 요소(Element)들 순회(traversing)하는 방법을 캡슐화한 객체
- 루아의 반복자는 '다음(next)' 요소를 반환하는 보통의 함수로 표현

```
function list_iter (t)
  local i = 0;   local n = table.getn(t)

  return function ()          -- 익명 함수
           i = i + 1
           if i <= n then
             return t[i]
           end
         end
end

t = {10, 20, 30}
iter = list_iter(t)           -- 반복자 생성
while true do
  local element = iter()    -- 반복자 호출
  if element == nil then
    break
  end
  print(element)
end
```


- 반복자와 for 문을 사용한 간단한 방법
  for 문은 nil을 반환할 때까지 순환

```
t = {10, 20, 30}
for element in list_iter(t) do
  print(element)
end
```

- 입력된 파일에서 모든 단어를 훑는 반복자

```
function allwords ()
  local line = io.read()                          -- 현재 줄
  local pos = 1                                    -- 현재 줄에서 현재 위치
  return function ()                               -- 반복자 함수
  while line do                                    -- 읽을 줄이 남아 있는 동안 반복
    local s, e = string.find(line, "%w+", pos)
    if s then                                      -- 단어를 발견 했는가?
    pos = e + 1                                    -- 이 단어 이후의 다음 위치
    return string.sub(line, s, e)                  -- 단어 반환
    else
    line = io.read()                               -- 단어 발견 실패. 다음 줄 시도
    pos = 1                                        -- 처음 위치부터 재시작
    end
  end
  return nil                                       -- 읽을 줄이 더이상 없음. 종료.
  end
end

-- for 문으로 allwords() 함수 사용
for word in allwords() do
  print(word)
end
```

- 반복문 마다 클로저를 생성하면 Overhead 발생 가능
- 루아의 Generic for 문은 반복 상태를 유지하도록 지정할 수 있음
  -->반복자 함수를 보관 Overhead를 줄임

- Syntax

```
for <변수 목록> in <수식 목록> do
   <본체>
End
```

- 변수 목록(variable name list): 한개 이상의 변수 이름을 쉼표로 구분

- 수식 목록(expression list): 한 개 이상의 수식들을 쉼표로 구분. 수식목록은 3개의 결과 값으로 조정

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

for line in io.lines() do
  io.write(line, '₩n')
end

for var_1, ..., var_n in exp-list do block end
-- 이것은 다음과 같이 대응됨
do
  local _f, _s, _var = explist
  while true do
  local var_1, ... , var_n = _f(_s, _var)
  _var = var_1
  if _var == nil then break end
    block
  end
end
```

- 다중 반복문에서 동일한 무상태 반복자를 사용해서 새로운 클로저의 생성을 피함

- ipairs() 함수는 stateless iterator 사용

```
a = {"one", "two", "three"}
for i, v in ipairs(a) do
  print(i, v)
end
```

- 루아로 구현한 ipairs() 함수와 pairs() 함수

```
function iter (a, i)
  i = i + 1
  local v = a[i]
  if v then
  return i, v
  end
end

function ipairs (a)
  return iter, a, 0
end

function pairs (t)
  return next, t, nil
end
```

- next(t, nil)은 첫 번째 키-값 쌍을 반환. 다음 결과 쌍이 없으면 nil 반환

- 몇몇 개발자들은 이점을 이용 pairs보다 next를 직접 사용

```
for k, v in next, t do
  ...
end
```

- 여러 제어와 많은 상태 값을 필요로 하는 제어문은 테이블 보다 클로저를 사용하는 것이 바람직
- 가능하다면 언제나 for 변수들에 자신이 모든 상태 값을 담는 무 상태 반복자를 작성 --> 불가능하면 클로저 사용
- 클로저는 테이블을 사용한 반복자 보다 효율이 좋음
- 테이블 생성보다 클로저 생성이 부하가 적다.
- 테이블 필드에 접근하는 것보다 비지역 변수에 접근하는 것이 더 빠름

- for문을 가지고 있지 않았던 루아의 이전 버전에서 사용한 반복자
- 반복문을 작성하지 않는 대신, 반복 처리를 할 때 마다 반복자가 해야 할 일을 설명하는 함수를 인수로 지정한 후 반복자 호출
- 반복자 작성이 용이하지만 발생자 방식보다 융통성은 떨어짐.
- 발생자 방식은 병렬 반복을 2개 이상 할 수 있고, 반복문 본체 안에서 break 문과 return 문을 사용할 수 있음
- True iterator의 return 문은 익명 함수의 실행을 반환하는 것.

```
function allwords (f)
  for l in io.lines() do
    -- repeat for each word in the line
    for w in string.gfind(l, "%w+") do
      -- 함수 호출
      f(w)
    end
  end
end
```

8 컴파일, 실행 오류 검사

- 루아는 중간 형태로 컴파일 한 후에 실행

- 컴파일 함수:
  - dofile() : 스크립트 파일 컴파일. 오류가 있으면 실행 중지. loadfile() + assert()
  - loadfile(): 스크립트 파일 컴파일. 오류가 있어도 실행을 계속, 오류 코드 반환
  - loadstring(): 문자열로 구성된 스크립트 컴파일. 호출 마다 문자열을 새로 컴파일
  - loadlib() : 주어진 라이브러리를 로드한 후 루아에 연결

```
-- loadfile을 사용한 dofile 구현
function dofile (filename)
  local f = assert(loadfile(filename))
  return f()
end

-- loadstring
f = loadstring("i = i + 1")  -- f는 i=i+1을 실행하는 함수가 됨

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


```
function foo ()
  ...
  if unexpected_condition then error() end
  ...
  print(a[i])     -- 잠재적 에러: a가 테이블이 아닐 수 있음
  ...
end

if pcall(foo) then
  -- 오류 없음
  ...
else
  -- 오류에 대한 처리
  ...
end

-- 익명 함수를 사용한 pcall
if pcall(function ()
                <보호 코드>
            end) then
                <정규 코드>
else
  <오류 처리 코드>
end

-- debug.traceback
print(debug.traceback())
```

- 대부분의 오류처리는 애플리케이션에서 처리
- assert(): 첫 번째 인수가 거짓(false 또는 nil)이면 오류를 발생시킴
- pcall(): protected call 함수는 보호 모드를 사용하여 함수가 실행 되는 동안 발생한 오류를 찾아냄
- xpcall(): 오류에 대한 역추적 정보를 제공하는 함수
- debug.debug() : 디버그용 함수
- debug.traceback(): 실행의 역추적 상황을 얻음

9 Coroutine

- 코루틴:
  - 별도의 스택과 지역 변수들을 가진 하나의 실행 줄(line of execution)이며 자신만의 명령 포인터를 가짐
  - 일종의 비선점형 멀티 스레드 기능
  - 루아의 스레드

- 스레드와 차이:
  - 코루틴을 사용하는 프로그램은 여러 코루틴 중에서 단 하나만 실행
  - 코루틴 스스로가 일지 중지(suspend) 요청 때만 실행을 중지

- 대칭형 코루틴:
  - 함수 한개만 사용하여 코루틴의 제어를 다른 코루틴으로 넘기는 것

- 비대칭형 코루틴(세미 코루틴):
  - 실행을 일시 정지하는 함수와 일지 정지된 함수를 재개하는 함수가 별도로 있는 것

- 루아는 완전한 비대칭형 코루틴
  - resume(), yield() 함수만으로 제어

- 모든 코루틴 관련 함수들은 coroutine 테이블에 묶어둠
- 코루틴 4가지 상태: 일시 정지(suspend), 실행(running), 죽음(dead), 정상(normal)
- 코루틴 제어 함수: create, status, resume, yield
- 코루틴 생성: coroutine.create(). 생성시 일시 정지 상태

```
co = coroutine.create(function ()
    print("hi")
end)
print(co)   --> thread: 0x8071d98
```

- 코루틴 상태: coroutine.status()

```
print(coroutine.status(co))   --> suspended
```

- 코루틴 실행(재실행): coroutine.resume(). 죽은 상태(dead)이면 실행 불가

```
print(coroutine.resume(co))   --> false   cannot resume dead coroutine
```

- 코루틴 일시정지: yield(). resume 될 때까지 대기

```
co = coroutine.create( function ()
                         for i=1, 10 do
                           print("co", i)
                           coroutine.yield()
                         end
                       end
                     )

coroutine.resume(co)            --> co   1
print(coroutine.status(co))     --> suspended

coroutine.resume(co)            --> co    2
coroutine.resume(co)            --> co    3

...
coroutine.resume(co)            --> co    10
coroutine.resume(co)            -- prints nothing

print(coroutine.resume(co))     --> false    cannot resume dead coroutine
```

- 주의) resume()은 보호 모드로 실행되어 오류는 resume() 호출 결과로 확인해야 함

- create, resume 에서의 인수 전달
- 코루틴은 resume, yield 함수끼리 데이터 교환 가능

```
co = coroutine.create(function (a,b,c)
      print("co", a,b,c)
    end)
coroutine.resume(co, 1, 2, 3)    --> co  1  2  3
```

- resume은 오류가 없다는 true와 대응하는 yield에서 넘기 모든 인수 값을 반환

```
co = coroutine.create(function (a,b)
       coroutine.yield(a + b, a - b, a * b)
     end)
print(coroutine.resume(co, 20, 10))  --> true  30  10  200
```

- yield는 대응 하는 resume으로 전달한 모든 추가 인수 반환

```
co = coroutine.create (function ()
                         print("co", coroutine.yield())
                       end)

coroutine.resume(co)
coroutine.resume(co, 4, 5)     --> co  4  5
```

- 코루틴 실행이 끝나면 코루틴 주 함수가 반환한 모든 값은 대응하는 resume으로 전달

```
co = coroutine.create(function ()
      return 6, 7
    end)
print(coroutine.resume(co))   --> true  6  7
```


- 생산자 소비자 문제

```
function producer ()
  while true do
  local x = io.read()       -- 새로운 값 생산
  send(x)                   -- 소비자로 전송
  end
end
function consumer ()
  while true do
  local x = receive()       -- 생산자로부터 수신
  io.write(x, "₩n")         -- 새로운 값 소비
  end
end
```

- 생산사 소비자 문제를 코루틴으로 변경

```
function receive ()
  local status, value = coroutine.resume(producer)
  return value
end

function send (x)
  coroutine.yield(x)
end

producer = coroutine.create(
  function ()
  while true do
  local x = io.read()       -- 새로운 값 생성
    send(x)
  end
  end)
```


- 반복문의 반복자는 값을 생산하고 반복문 본체가 그 값을 소비하는 생산자-소비자 패턴 형태 --> 코루틴으로 작성 가능

```
-- 주어진 원소로 모든 순열 만들기
function permgen (a, n)
  if n == 0 then
  printResult(a)
  else
  for i=1,n do
    -- put i-th element as the last one
    a[n], a[i] = a[i], a[n]

    -- generate all permutations of the other elements
    permgen(a, n - 1)

    -- restore i-th element
    a[n], a[i] = a[i], a[n]
  end
  end
end

 function printResult (a)
  for i,v in ipairs(a) do
  io.write(v, " ")
  end
  io.write("₩n")
end

permgen ({1,2,3,4}, 4)
--> 2 3 4 1
...
--> 1 2 3 4
```

```
-- 반복자로 변경
function permgen (a, n)
  if n == 0 then
  coroutine.yield(a)
  else
  ...

function perm (a)
  local n = table.getn(a)
  local co = coroutine.create(function () permgen(a, n) end)
  return function ()   -- 반복자
  local code, res = coroutine.resume(co)
  return res
  end
end

-- 적용
for p in perm{"a", "b", "c"} do
  printResult(p)
end
-->b c a
...
```

- wrap():
  - create 처럼 새로운 코루틴을 만들지만 자신을 반환 안하고 resume() 함수를 반환.
  - 첫 결과로 오류 코드를 반환 안함.
  - create 보다 간단한 대신 오류나 상태를 확인할 방법이 없음.

```
function perm (a)
  local n = table.getn(a)
  return coroutine.wrap(function () permgen(a, n) end)
end
```