Game Programming with LUA

afewhee@gmail.com

- The Language
- Types and Values
- Expressions
- Statements

1 언어 (Language)

- 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 안됨
```

- 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.

1.2 어휘 규정 (Lexical Conventions)

- 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 & Value)

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

- 바인딩: 변수 또는 함수에 속성을 부여하는 것

- 바인딩 Time:
  - 컴파일 전 pre-processor
  - 컴파일: Compiler
  - Linking: extern 변수
  - Loading: DLL 관련 변수
  - Runtime: 프로그램 실행 중

- 정적 바인딩: 프로그램 실행 전 바인딩

- 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
  - 실행 중에서도 변수를 사용하지 않으면 타입이 미리 결정되지 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 등의 자원 해제 때 '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
```


2.5 테이블 (table)

- 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
- table 길이: # 또는 table.getn(v)

- 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)

- 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
```

- 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
```

- For 문 종류
  - Numeric (수치) for 문
  - Generic (일반) for 문

- Numeric (수치) for 문 Statements

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

Numeric (수치) for 문

- 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
```

- Generic (일반) for 문

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


-- 색인, 값  출력
for i,v in ipairs(days) do
        print(i, v)
end

-- 색인 만 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
- return 문은 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
```