Témalabor dokumentáció
Változatos programozási nyelvek és környezetek megismerése

Szalai Dávid
AGGRW2

konzulens:
Kövesdán Gábor

2018. december 5.

Tartalomjegyzék

1. fejezet

Bevezetés

1.1. Téma leírása

A szoftverfejlesztésben általában egy fő platformra specializálódunk, és ebben fejlesztünk. Ez praktikus döntés, hiszen a terület olyan gyorsan fejlődik, hogy nem lehet minden nyelven és platformon mély és naprakész tudást fenntartani. Emiatt azonban a fejlesztők hajlamosak nagyon egyoldalúan gondolkodni, csak a saját maguk által preferált környezet kínálta megoldásokat, technikákat használni és elfogadni.

A téma lehetőséget kínál az érdeklődő hallgatóknak, hogy több, jelentősen különböző nyelvekbe is belekóstoljanak, és ezáltal jobban megismerjék a manapság elérhető programozási nyelvek kínálta technikákat és lehetőségeket. A választott nyelveken olyan programokat kell elkészíteni, amelyek elég egyszerűek ahhoz, hogy legyen idő több programozási nyelven is megírni őket, ugyanakkor elég bonyolultak is legyenek, hogy a választott nyelvek eszköztárát alkalmazni lehessen rajtuk. Ilyen lehet például a black jack kártyajáték vagy a game of life.

A munka kimenete, hogy a hallgató bár nem feltétlen lesz "profi" egyik nyelvben sem, egy magas szintű rálátással rendelkezik majd az egyes nyelvekre és platformokra. Ez szolgálhat alapul a későbbi választáshoz, illetve akár konklúziók is megfogalmazhatók, hogy melyik nyelv milyen problémákban erős.

Javsolt nyelvek: Java, Kotlin, Python, JavaScript, Scala, Haskell, Erlang, Lua. A feladatnak az is része, hogy a használható fejlesztőkörnyezeteket és library-kat is feltérképezzük, ezért a feladatokat TDD megközelítéssel kell elvégezni.

A téma továbbvihető későbbi félévekre, sőt, igényes kidolgozása több félévet igényel.

1.2. A projekt célja

Alapvetően azért választottam ezt a témát, mert szerettem volna kicsit elrugaszkodni az eddig használt nyelvek világától és megismerni más, akár típusban eltérő, nyelveket is. A másik oka, hogy ezt a témát választottam, pedig az, hogy több nyelv megismerésével talán konkretizálódik, hogy mivel szeretnék foglalkozni a jövőben. Úgy gondolom, ha egy programozási nyelv megtetszik, akkor sokkal könnyebb témát keresni hozzá, hiszen ismertek a lehetőségei.

A célom pedig az volt, hogy tágítsam a látókörömet és ezáltal talán kialakul majd egy kép bennem, hogy milyen irányban induljak el a jövőben. A témalaborig csupán csak azokkal a nyelvekkel ismerkedtem és foglalkoztam, amiket az egyetemi évek alatt oktattak. Fontos volt számomra, hogy tágítsam a látókörömet. Persze már a választásomkor tudtam, hogy egy félév nem lesz elegendő, hogy megismerjek minden nyelvet, amit szeretnék. De úgy vélem, hogy kiválasztva a két legszimpatikusabbat, azért jelentős tapasztalatokkal gazdagodhatok, akkor is, ha tudom, hogy még rengeteg más nyelv van, amit még érdemes kipróbálni.

Ahogy a témaleírás1.1 is írja, ez a téma tipikusan továbbvihető a későbbi félévekre, hiszen nagy rész teljesen önálló munka. Én úgy érzem a lényegét mégis átadta, hiszen bár sok különböző nyelv van, azért az egyes típusokban sok a hasonlóság (gondolok itt például az OO vagy a szkript nyelvek világára, akár a funkcionális programozásra). Alapvetően ez egy remek téma és lehetőség volt, hogy betekintést nyerjek más típusú nyelvek világába és megismerhessem sajátosságaikat.

1.3. Választott nyelvek

Már a témalabor választásakor elgondolkoztam, hogy melyik nyelvek legyen azok, amelyekkel egy féléven át foglalkozni szeretnék. Több elképzelésem is volt, de nem akartam túlzásba esni. Ezért két szempont szerint elkezdtem szortírozni a lehetőségeket.

Először is, mindenképpen szerettem volna választani egy OO nyelvet, amit még sosem használtam. A választásom pedig azért a Kotlin3 lett. Ennek több oka is volt. A barátaim közül többen is javasolták, hogy válasszam ezt, mert ők egy nyári projekt alkalmával kipróbálták és rendkívül tetszett nekik. Ösztönöztek, hogy próbáljam ki én is. A másik ok, pedig egy kis utánajárás következtében alakult ki. Pár bemutatóvideó után nekem is megtetszett ez a nyelv, mert bár Java alapokra van helyezve, mégis szinte minden hibáját kijavították benne. Sőt, ha kicsit pongyolán akarok fogalmazni, akkor összegyúrták a Java és a C# nyelvet és az ő gyermekük lenne a Kotlin. Tehát ezzel mindenképp meggyőzött, hogy ez legyen az egyik választott nyelvem.

A másik szempontom pedig, hogy válasszak egy nem OO nyelvet. Először még nem tudtam eldönteni, hogy szkript vagy deklaratív nyelvet válasszak, így csak egy sorrendet tudtam kialakítani, hogy előbb lenne egy szkript nyelv és aztán egy deklaratív.

A szkript nyelvek közül nehéz volt választani. Bár szerencsére sok a hasonlóság, azért mindegyiknek megvannak a maga sajátosságai. Ilyenkor persze mindenkinek először a JavaScript jut eszébe és én épp ezért gondolkodtam valami másban. JavaScript betekintőt a félév során egy másik tantárgyból úgyis kapunk, ezért is gondoltam úgy, hogy eltérek a "mainstream" választástól. Épp emiatt a második nyelvemnek a Lua-t2 választottam.

Volt egy másik oka is, amiért ez lett a választás. Gyakran játszok ugyanis egy MMORPG játékkal, amelyben használhatok külső, kisegítő eszközöket (ún. Addonokat), és mindig is érdekelt, hogy ezeket hogyan csinálják, mert én is szeretnék csinálni számomra testre szabottakat vagy akár a meglévőket módosítani. Amikor utánajártam, hogy ezeket az Addonokat Lua-ban írják, akkor gondoltam úgy, hogy szeretnék foglalkozni ezzel a nyelvvel, ha lesz rá időm. És ezért esett a választás erre a nyelvre.

Mindazok a nyelvek, amiket nem választottam ki, de a munkám során vagy korábbi évek alatt előkerültek (Java, C, C++, C#, JavaScript, Pascal, Python), szeretném majd (ha tudom) referenciaként vagy akár "harmadik szemszögként" felhasználni. Így a nyelvek bemutatását még több perspektívával tudom ellátni és bemutatni, hogy mennyi apróbb különbség/hasonlóság van a nyelvekben.

Deklaratív nyelvet azért is raktam a végére, mivel a 7. félévben lehetőség lesz egy elágazó tantárgy kereteiben tanulni róla. Mindemellett úgy gondoltam, ha jut rá időm, szeretném kipróbálni a Haskell-t. Sajnos azonban ez a terv nem tudott megvalósulni a félév alatt.

1.4. A projektek bemumtatása

Először bemutatom a választott nyelveket vázlatosan. Az egyes példánál a projektekből idézek, illetve megpróbálok minél több referenciát felhozni, hogy más nyelvekben milyen hasonlóságok és különbségek vannak.. Majd ezután egy összehasonlítást mutatok be mind a BlackJack, mind a GameOfLife projektekkel kapcsolatban, melyben a két nyelv különbségeit és hasonlóságait mérem össze.

2. fejezet

A Lua nyelv és sajátosságai

2.1. Kicsit a Lua-ról

A Lua egy programozási nyelv, amit 1993-ban Roberto Ierusalimschy, Luiz Henrique de Figueiredo és Waldemar Celes fejlesztett a Pontifical Catholic University of Rio de Janeiro egyetemen Brazíliában. A Lua portugálul holdat jelent és a helyes kiejtése "LOO-ah".

A Lua előnyei a bővíthetőség, egyszerűség, hatékonyság és hordozhatóság. Könnyű hozzá új metódusokat/funkciókat és modulokat. Gyakori szkript nyelv játék programokhoz. Az egyszerűsége ellenére a Lua nagyon erős ún. "multi-paradigm" programozási nyelv. Ezálzal több stípust is támogat, legyen az imperatív, funkcionális vagy objektum-orientált. Emellett a Lua-ban nem kell foglalkozni a memóriakezeléssel, ugyanis egy nagyon jó inkrementális garbage collector-ral rendelkezik. A Lua Pascal szerű szintaxist követ.

A másik nagy előnye a Lua-nak, hogy rendkívül gyors. A virtuális gép, amin a Lua 5.1-t használnak, az egyik leggyorsabb az olyan programoknál, amik szkript nyelveket használnak. Van még egy "just-in-time" fordítja is (a Lua-t a számítógép natív gépi kódjára fordítja le miközben fut a program), amely az x86 architektúrák számára lett tervezve és emiatt még gyorsabb.

2.2. Adattípusok és változók

2.2.1. Változók

A változó jelentése itt is annyit tesz, hogy egy érték tárolható benne, amire a neve segítségével lehet hivatkozni pl:

```
done = false
```

Az értékadás hasonló, mint a legtöbb nyelv esetében. Változó név szinte lehet bármi, de hasznos, ha beszédes nevet kapnak. Vannak azonban kulcs szavak a Lua-ban is, amelyeket nem szabad használni:

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

Ami még érdekes (bár ez majd a többi rész után egyértelmű lesz), hogy ezeken a kulcsszavakon kívül a Lua nem használ mást. Nagyon érdekes, hogy ezekkel szinte mindent le tudunk írni. Emellett a változók kis és nagybetű érzékenyek, ami szintén hasznos tulajdonság.

2.2.2. Típusok

A Lua-ban alapértelmezetten 7 típus van (mondhatni ezek a beépített típusok): **nil, number, string, boolean, table, function és thread**. Szintén érdekesség, hogy ezzel a 7 típussal szinte mindent le tud fedni. Gyorsan vegyük sorra az egyes típusokat és jellemzőiket.

nil

A nil típus lényegében azt jelenti, hogy valaminek nincs hasznos értéke. Például, ha egy változó nincs inicializálva, akkor az alapértelmezett típusa nil.

```
...
 person
 print(typeof(person)) --> nil
 person = 5
 print(typeof(person)) --> number
...
```

Egy másik eset például egy függvény meghívásánál, ha nem adunk át elég paramétert egy függvénynek (pl: 3 paramétert vár és csak 2-t kap), akkor az a változó nil típusú lesz. Ez a típus tényleg a "különleges állatfajok" világába tartozik, ugyanis nem egyenértékű például a Java vagy a C++ null típusával. Ugyanakkor Nem is tekinthető default value-nak, mivel a nil típus csak a nil értéket értelmezi.

Talán a JavaScript tudná a legjobban körülírni. Ott ugyanis egy közeli fogalom létezik: **undefined**. Az undefined az inicializálatlan változót jelenti. A nil vegyíti a null és az undefined fogalmát és használatát (mondhatni ez a két fogalom nem vált ketté a Lua-ban).

number

A number is szintén egy érdekesen viselkedő típus. Míg más nyelvekben, mint például a C++ vagy a Java külön típusok vannak az egész és a tört számokra, addig a Lua-ban ezek mondhatni egyesítve vannak a number típusban. Ha kell Integer-ként viselkedik, ha kell, akkor Double vagy Float-ként. Alapértelmezésben minden szám Double ($-1.79 * 10^{308}$ és $1.79 * 10^{308}$ közötti értékek).

Akárcsak a Java-ban vagy a C#-ban, itt is lehetőség van szövegből számot csinálni. Erre való a **tonumber(str)** függvény.

```
...
print(tonumber("5")) --> 5
print(tonumber("-100")) --> -100
print(tonumber("0.5")) --> 0.5
print(tonumber("-3e5")) --> -300000
...
```

string

A string teljesen hasonlóan viselkedik a Java vagy a C# stringjéhez képest. Azokhoz hasonlóan itt is "immutable"-ö, azaz módosíthatatlanok. Itt is a Lua a szövegeket egy string pool-ban tárolja. A változó sosem tárolja az értéket, csupán a referenciát a string poolra. Az escape karakterek (pl

a
a,
n vagy a
t) teljesen ugyanúgy működnek, mint más nyelvekben. Talán ez a típus a legsablonosabb mind közül, hisz a Lua-ban nincs plusz feladata.

Fontos még kitérnem egy tipikus tulajdonságra, amely sok nyelvbe eltérő: két string összeadása. Maga az a tény, hogy egy új string jön létre, az majdnem minden nyelvben ugyanaz (a Lua-ban is), viszont az összeadás módja eltér. Nem véletlenül használom az 'összeadás' kifejezést, hiszen a legtöbb nyelvben (pl. Java, Kotlin, C#) két stringet az összeadás operátorral kötünk össze. Lua-ban viszont ez eltér. Itt érdekes módon nem szokás az összeadás operátort használni. Helyette a **..** operátort használja:

```
str = "hello".."world"
print(str) --> hello world
```

Valamint fontos megjegyezni, hogy a Lua-ban ez az operátor impliciten átalakítja a nem string típusú változókat string-gé (ha lehet, pl. number-t igen). Így mindenféle probléma nélkül lefut a következő kód is:

```
str = "hello␣"..1.."␣world"
print(str) --> hello 1 world
```

Ez igen kényelmes és hasznos funkció, amely sok nyelvben gyakran plusz függvények meghívásával tudunk elérni. Illetve komplexebb típusoknál (kifejezetten table objektumokra értem), ha felül van definiálva a **__tostring** operátor (erről a 2.9.1. fejezetben fogok részletesebben beszélni.) , akkor automatikusan átalakítja string típusúvá.

Utoljára még megemlíteném a hossz operátort, amely Lua-ban kicsit eltér a más nyelvekben megszokottaktól. Java-ban vagy C++-ban egy külön függvényt definiálunk erre és nem operátort. Míg C#-ban property segítségével tudjuk lekérdezni egy string hosszát. Lua-ban erre a **#** operátort használjuk.

boolean

Érdemben teljesen hasonlóan működik ez a típus, mint minden más nyelvben. Két értéke lehet: a **true** és a **false**.

function

A function típus követi a szkript nyelvek közös vonásait. Függvény kétféleképpen lehet megadni:

```
function foo(arg1, arg2)
        print(arg1)
        print(arg2)
end
```

vagy

```
foo = function(arg1, arg2)
        print(arg1)
        print(arg2)
end
```

Igazából az első példa csak egy egyszerűbb leírása a másodiknak. Valamint az első példa alapján azt feltételeznénk, hogy van egy függvény, amit foo-nak hívnak. Valójában a második példa alapján jól látszik hogy egy változót hozunk létre, amely tárol egy függvényt. Magának a függvénynek nincs neve, mert a Lua-ban az összes függvény anonymus function.

```
foo = function(arg1, arg2)
        print(arg1)
        print(arg2)
end

bar = foo
```

Így már két változó mutat ugyanarra a függvényre. De fontos, hogy a függvény soha nem másolódik, csupán a referencia (az ilyen dolgot megszokottak pl. a Java és a C# világában, hiszen ott csak referencia típusú átadás van, de a C és C++-ban előfordul érték szerinti átadás). A függvények viselkedéséről még a későbbiekben kitérek.

thread

A thread név ennek a típusnak kicsit csalóka lehet, ugyanis merőben eltér a Java vagy a C# Thread típusaitól. Lua-ban ugyanis nem valósul meg a többszálúság... Egy sokkal jobb megnevezés a **corutine** (cooperating routine). Egy corutine tulajdonképpen egy függvénynek felel meg 1 különbséggel Két plusz tulajdonsággal rendelkezik: tudnak **yield**-elni és vissza lehet tölteni őket. A yield függvény nagyjából a Java-ban a Sleep metódusnak felel meg, míg a visszatöltés automatikusan megtörténik, azzal nem kell foglalkozni.

A különbségre a magyarázat már a névből is kikövetkeztethető valamint a számítógépek ütemezőivel lehet jellemezni. Kétféle típusú lehet egy ütemező: preemptív, azaz elveheti egy száltól a futás jogát (ezzel megszakítva a jelenlegi futását), vagy kooperatív, ebben az esetben az ütemezőnek nincs joga elvenni a futás jogát.

Igazából a Lua-ban az utóbbi valósul meg, tehát nem igazán beszélhetünk többszálúságról, hiszen a szálak nem küzdenek a processzorért, ugyanakkor, ha épp nincs szükség, átadhatják a futás jogát (yield) és majd visszatérnek, ha nincs más várakozó. A corutine-ok menedzselésére a Lua-nak van egy beépített könyvtára: **corutine** névvel. Egy példa segítségével megpróbálom gyorsan bemutatni, hogy mik is ezek a corutine-ok:

```
foo = function()
        for i = 1, math.huge do
                corutine.yield(i)
        end
end

local co = corutine.wrap(foo)
print(co()) --> 1
print(co()) --> 2
print(co()) --> 3
```

Ismételten hasznos megjegyezni, hogy nincs többszálúság, az a fogalom, hogy a főprogram szála, nem létezik. Ugyanakkor meg lehet valósítani többszálúsági problémákat, mint például egy consumer-producer problémát, bár a Lua nem erre lett kitalálva. Azonban jó látni, hogy ha kell, erre is nyújt egyfajta lehetőséget a nyelv (sajnos elég gyorsan túlkomplikált és nehezen érthető lesz miattuk a kód).

table

A végére marad a Lua talán legérdekesebb és leghasznosabb, valamint legfontosabb típusa. A table típus segítségével tudunk implementálni bármiféle adatstruktúrát. Az alapokat itt felvázolom, de a későbbi részekben kitérek még a sajátos viselkedésére.

Nagyon leegyszerűsítve a table egy kulcs-érték párokat tároló tömb. Nagyon hasonlít a map-re és az array-re, de valójában egyik sem és mindkettő egyszerre. Rendkívül különleges már csak az adattárolási lehetőségei.

```
table = {"a", 7, false, -1}
```

Ha egy table-t szeretnénk megadni, akkor azt evvel = { } a módon adhatjuk meg. A példában jól látható, hogy nem számít mit tárolunk egy table-ben. Ugyanakkor működik természeten az index operátor:

```
print(table[1]) --> a
print(table[3]) --> false
print(table[5]) --> nil
```

Itt két érdekesség figyelhető meg. Az első, hogy a számozás nem 0-tól, hanem 1-től indul. Ez tipikusan zavaró tud lenni a C, C++, Java vagy akár Kotlin nyelvek után, de nem egyedi (pl.: FORTRAN, SASL, MATLAB, Erlang). A másik, hogy míg csak 4 elem volt a táblában meg tudtam hívni az 5. indexűt. Mivel értelmes értéket nem tároltam benne így nil típusú az az érték, amit visszaadott és az értéke is nil.

Ami viszont érdekesség, az a következő példán látszik:

```
table[1000] = "z"
print(table[1000]) --> z
```

Ekkor azt gondolnánk, hogy a table újraméretezte magát, de valójában ben ez történt. És itt mutatkozik meg a table szépsége és sokoldalúsága. Ekkor ugyanis a table szétválik két részre. Egy iterálható és egy tisztán kulcs-érték párokon alapuló részre. Hogy mit is értek kulcs-érték párokon, azt a következő példa megmutatja:

```
table = { key1 = "a", ["egy␣kulcs␣szokozokkel"] = "b",
          [5] = 1, [print] = false }
```

A táblám tartalmaz egy key1 kulcsot, aminek az értéke 'a', ez még szinte alap is, de a többi példa már érdekesebb. A [5] nem ugyanaz, mintha egy szöveg lenne. Így tudok explicit indexet létrehozni. De akár függvénynevet (azaz egy változó nevet) is megadhatok kulcsként.

```
print(table["key1"]) --> a
print(table.key1) --> a
print(table[print]) --> false
```

A kulcs-érték párokat a fenti példán látva kétféle módon tudom meghívni. Míg az iterálható elemeket a szokásos módon. Visszatérve a korábbi példára, ahol az 1000-ik indexet adtam meg, már jobban érthető, hogy hogyan is lehetséges ez és hogy valójában mit is jelent.

Az iterálás lehetősége ezekben a table típusú objektumokban kétféle módon lehetséges. Az első a jól megszokott index szerinti iterálás. Csak ezzel van egy kis probléma. Ugyanis, ha az indexek valahogy így néznek ki: 1,2,3,5,7,1000... akkor mi is a szabály. Ez pedig nagyon frappánsan van kitalálva a Lua-ban.

Amíg tud egyesével iterálni, azaz vannak indexek, amiken végig tud menni, addig azokon az elemeken végigmegy, amíg tud. Ha azt akarom, hogy az első 5 elemet iterálja, akkor a 4. indexűnek is értéket kell adni. Ekkor már el tudok iterálni az 5. indexűig is. Ekkor felmerül a kérdés, hogy mi lesz a többi elemmel? Például azokkal, amik kulcs-értékként vannak tárolva. Mivel a table hátterében igazából egy hash map áll, így azokon az elemeken is végig tudunk menni. Ez a példa bemutatja először az index majd a hash map szerinti iterálást:

```
table = {1, 2, 3, key1 = "a", ["egy␣kulcs␣szokozokkel"] = "b",
          [5] = 1, [print] = false, [1000] = "k" }
```

```
for k,v in ipairs(table) do
        print(k, v) --> [1] 1, [2] 2, [3] 3
end

for k,v in pairs(table) do
        print(k, v) --> [key1] a, [2] = 2, ...
end
```

Az első for ciklusnál jól látható, hogy csak az indexelt elemek jelennek meg. A többit így nem tudjuk kilistázni. Ahhoz a másik for ciklusra van szükség. A kettő szinte teljesen megegyezik 1 különbséggel: az elsőben ipairs, míg a másikban pairs a függény neve. A kettő lényegében, annyiban különbözik, hogy az ipairs az iterálható elemek tömbjén megy végig, míg a sima pairs a hash mapen. Azért nem írtam a másodikhoz a teljes kimenetet, mert ez előre nem meghatározható. Ugyanis ez a hash sorrendjében járja be a table elemeit és az a sorrend nem feltétlenül egyezik meg az én deklarációmmal.

Az értékadás hasonlóan működik, akárcsak a stringeknél. Itt is referencia szerinti átadás történik. Fontos megjegyezni, hogy ettől még a Lua-ban nem csak referencia szerinti átadás van. Például a number típusú változók értékeit másolja. Mondhatnánk, hogy a number primitív típus, de ezt már megtárgyaltuk, hogy ez nem teljesen lenne igaz.

2.2.3. Összegzés

Annyit még mindenképpen meg kell említenem, hogy az egyes változók a nincsenek egy konkrét típushoz kötve. Ez azt jelenti, hogy a Lua egy ”dynamically typed” nyelv. Ellentétben a C, C++, Java vagy Kotlin nyelvektől, amik erősen típusosak. A dinamikus típusosság jellemző a szkript nyelvekre, amely rendkívül nagy rugalmasságot biztosít, ugyanakkor megvannak a hátulütői. Ugyanis nagyon könnyű elveszni, hogy egy változó mögött mi is áll (egyen talán áll-e valami). Szerencsére a Lua-nak van egy egész hasznos error generálója, amely értelmes üzenetek tud adni és egy backtrace-t is nyújt.

2.3. Kifejezések

A kifejezés nagyjából annyit jelent, hogy operátorokat használ különböző értékeken és kiértékeli egy értékké. Lua-ban bárhol lehet használni kifejezéseket, ahol a Lua egy értéket vár. Pl. változó értékadásakor vagy függvény paramétereként.

2.3.1. Aritmetikai operátorok

Lua-ban is megvannak az alap aritmetikai operátorok: `+` összeadás, `-` kivonás, `*` szorzás, `^` hatványozás, `/` osztás és a `%` moduló operátorok. Ezek csak number2.2.2 típusú változókkal és értékekkel működnek. Ezen kívül még létezik a negációs operátor, amely szinte teljesen megegyezik a mínusz operátorral, de ennek csak egy paramétere lehet:

```
a = 5
print(0 - a) --> -5, normal minusz
print(-a) --> -5, unary
```

Elsőre ennek nincs jelentősége, de a precedenciáknál fontos különbségnek számít.

2.3.2. Relációs operátorok

Lua-ban a relációs operátorok nagyjából teljesen megegyeznek más nyelvek operátoraival: `==` egyenlőség, `<=` kisebb-egyenlő, `>` nagyobb, stb. Ugyanúgy boolean2.2.2 értéket adnak vissza. Egye-

dül a nem egyenlő operátor tér el Java-tól vagy a C, C++-tól megszokottaktól (ami a `!=` forma). Itt ugyanis `~=` a jelölés. De ez csak jelölésben tér el, használatban teljesen azonos.

2.3.3. Logikai operátorok

Az operátorok harmadik típusába három logikai operátor tartozik: `and, or` és `not`. Ami érdekes, hogy az `and` és az `or` operátor bármilyen típus között működik és bármilyen típust vissza tud adni, míg a `not` csak boolean típust ad vissza. Ezek az operátorok minden értékhez igazat rendelnek, kivéve a false-t és a nil-t. Ez a szkript nyelveknél gyakori, de abban általában eltérnek, hogy melyik értéket tekintik false-nak és melyiket true-nak. Pl a JavaScript-ben a 0 false-ként viselkedik és az üres string is, míg a Lua-ban ezek mind true-ként. De talán a legérdekesebb a 0 értelmezése, hiszen a legtöbb nyelv (min pl a C) gyakran kihasználja, hogy a 0-t false-nak tekinti.

Még egy érdekesség a "logikai rövidzár" fogalma. Azaz a logikai operátor második tagját nem értékeljük ki, ha nincs rá szükség. Ez az `and` operátornál azt jelenti, hogy ha az első értéke false, akkor a másikat már ki sem kell értékelni, hiszen a kifejezés biztos false lesz. Az `or` esetében, ha az első kiértékelés után true-val tér vissza, akkor szintén nem kell a másikkal foglalkozni, hiszen a kifejezés értéke biztos igaz lesz. Ez a legtöbb nyelvben ugyanúgy jelen van és gyakran ki is használják, a Lua-ban pedig különösen.

Az alapértelmezett érték (default value) fogalmát a legtöbb nyelv kihasználja. Ennek egy tipikus esete a függvények meghívásánál jön elő. A lényeg, hogy a függvény egyes paramétereinek értékét nem feltétlenül szükséges megadnunk vagy szinte minden esetben ugyanazok, emiatt hasznos dolog lenne, hogy egy alapértelmezett értéket be tudjunk nekik állítani, így a függvény meghívásánál ha nem szükséges, akkor nem kell megadni.

A legtöbb nyelv ezt támogatja, de megvalósítása gyakran eltérő. C++ és C# egyszerűen a változó mellé egy egyenlőségjel után odaírjuk az alapértéket, addig például a Java-ban erre nincs lehetőség. Ott function overload-ot használják ki, ami nagyjából abból áll, hogy meghívják a kevesebb paraméterű függvényt, ami továbbhívja a teljes paraméterlistával rendelkező alapfüggvényt, csak már kiegészítve a hiányzó paraméterek alapértékével. De vannak olyan nyelvek, mint például a C, ami nem támogatja ad default értéket függvényeknél. Ott külön függvényeket kell létrehozni minden esetre.

A Lua-ban alapértelmezett értéket a függvény fejlécében nem tudunk megadni, de ezzel ellentétben ki tudjuk használni a logikai operátorokat és az eddigi ismereteinket a függvények paramétereiről. Ha egy függvény paraméterének nem adunk értéket, akkor az attól még meghívódik. Ekkor felmerül a kérdés, hogy mi lesz annak a paraméternek az értéke. Erre használjuk tipikusan a nil típust. Azaz minden "undefined" paraméter értéke nil lesz. Ezt pedig a logikai operátor használja ki. Ugyanis a nil értéke false-nak minősül.

```
function new(value, color, number, hidden)
        local newCard {}
        newCard.value = value
        newCard.color = color
        newCard.number = number
        newCard.hidden = hidden or false
        return newCard
end

card = new("KING", "SPADES", 11)
```

Ebben az esetben a hidden értéke nil lesz. Amit az értékadásnál a logikai operátor kihasznál, így bármi is áll a jobb oldalon, az az érték fog beleíródni a table "hidden" attribútumába. Ha a függvényt 4 paraméterrel hívjuk meg, akkor pedig a logikai rövidzár miatt az átadott paraméter hívódik meg. Itt fontos megjegyezni, hogy egy kivételes esetre komolyan fel kell készíteni a

függvényt. Ha false értéket adunk át, jól látszik, hogy a rövidzár akár kellemetlenül is hathat az értékre. Ezt egyedül jó meggondolással lehet csak kiküszöbölni (azaz mit írjunk az kifejezés jobb oldalára).

2.4. Vezérlő struktúrák

2.4.1. Elágazások

A Lua elágazások és ciklusok terén eléggé szűkre szabott. Elágazásokból az if-else állítás értelmezett, de körülbelül ennyi. A gyakran használt switch-case megoldásra nincs külön típus, sajnos esle if-ek segítségével lehet csak megoldani. Mindazonáltal fontos megemlíteni a szintaxist.

```
i = 5

if i > 2 then
 --> do smoething
end

if i < 4 then
 --> if statement is true
else
 --> if false
end
```

Látható, hogy nem kapcsos zárójeleket használ, hanem `then - end` párokat. Valamint, ha a kifejezés nem összetett, akkor a feltétel zárójelezése is elhagyható.

Fontos különbség azonban, hogy az `else if` és az `elseif` nem egyezik meg. Ugyanis az `else if` egy új blokkot nyit, míg az `elseif` az eredetit használhatja:

```
i = 5

if i > 2 then
--> do smoething
end

if i < 4 then
-->  if statement is true
elseif i > 6 then
--> ha hamis
                        --> nincs 'end' tag
end

if i < 4 then
-->  if statement is true
else --> az 'else if' szebb alakja
        if i > 6 then
                --> ha hamis
        end --> kell 'end' tag
end
```

Így látható, hogy a switch-case megoldást a Lua az `elseif` segítségével tudja megvalósítani:

```
if(result == ResultType.WIN) then
        print("You win!")
        playerOdds = 2
        bankOdds = -1
        self.winCount = self.winCount + 1
elseif(result == ResultType.WINBYJACK) then
        print("BlackJack! You win!")
        playerOdds = 2.5
        bankOdds = -1.5
        self.winCount = self.winCount + 1
elseif(result == ResultType.TIE) then
        print("It's tie!")
        playerOdds = 1
        bankOdds = 1
elseif(result == ResultType.LOSE) then
        print("You lost!")
        playerOdds = 0
        bankOdds = 1
        self.loseCount = self.loseCount + 1
elseif(result == ResultType.LOSEBYJACK) then
        print("You lost!")
        playerOdds = -0.5
        bankOdds = 1.5
        self.loseCount = self.loseCount + 1
else print("error")
end
```

2.4.2. Ciklusok

A Lua-ban jelen van a három megszokott ciklus: a**for, while, do – while**. Habár az utóbbi szintaxisa kicsit eltér a megszokottól. Itt is, mint az elágazásoknál, kicsit rendhagyó a szintaxis a megszokott kacsos zárójelektől, ami tipikusan jellemző a C, C++, Java vagy akár a Kotlin világában.

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

Itt is felvehetünk egy ciklusváltozót, amely lokálisan lesz jelen a ciklus törzsében, akárcsak a megszokott módon pl. C++-ban. Itt is a második a megállási feltétel, míg a harmadik az iteráció nagysága. A zárójelezést a ciklusok esetében a **do – end** páros alkotja. Valamint fontos megjegyezni, hogy az elválasztások itt nem ;-t használnak, mint a legtöbb nyelvben (C, C++, Java), hanem csak sima vesszőt.

Így néz ki a tipikus for ciklus, de persze megannyi módon tudjuk változtatni. Alapesetben, ha egyesével iterálunk növekvő számokkal, akkor elhagyható a harmadik tag:

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

Emellett még egyszerűsíteni is lehet a megállási feltételen, ha a ciklusváltozó és a feltétel típusa megegyezik:

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

```
end
```

Legtöbbször persze a for ciklust arra használjuk, hogy egy listán/tömbön végig iteráljunk. Egy korábbi példában már utaltam arra, hogy ez lehetséges:

```
for key, value in pairs(table) do
        print(key, value)
end
```

Ez a példa tipikusan a ”foreach” megfelelője. A ”pairs” függvény visszaad egy kulcs-érték párokat tartalmazó table-t és az ”in” kulcsszó segítségével kulcs-érték párokat ad értékül a ”key” és ”value” változóknak. Ezt annyiszor teszi meg, ahány elemmel tér vissza a pairs függvény által visszakapott table. Azt, hogy a két értéknek hogyan adhat így értéket, arra a későbbiekben visszatérünk2.5.

A **while** ciklus teljesen hasonlóan viselkedik a C, C++ vagy a Java-s barátaihoz képest:

```
i = 1
while i <= 10 do
        print(i)
end
```

Azonban a **do** − **while** kicsit eltér. Lua-ban ugyanis a **repeat** − **until** kulcsszavakat használjuk. Ezen kívül a logika tejesen megegyezik a megszokott viselkedéssel:

```
i = 1
repeat
print(i)
until i > 10
```

Fontos megjegyezni, hogy itt nincsen `do -- end` blokk.

Még egy fontos dolog a ciklusokkal kapcsolatban, hogy értelmezzük a **break** fogalmát Lua-ban is. Teljesen hasonlóan viselkedik, mint a többi nyelvben: megszakítja a ciklust és kilép belőle az adott ponton:

```
table = {"a", "b", 5, 10, 100}
for k, v in pairs(table) do
        if(v == 5) then
                key = k
                break
        end
end
print(key) --> 3
```

Itt két dolgot érdemes megfigyelni. Az egyik, amely a legszembetűnőbb, hogy nincsenek pontosvesszők a sorok végén. Ez már korábban is előjött, de talán itt érzem már fontosnak megemlíteni, hogy miért is lehet elhagyni őket. Mivel a Lua egy szkript nyelv, a fordítása interpretálva történik, azaz sorról sorra (pontosabban kifejezésről kifejezésre) futtat. Így egy sor vége elég jelzés , nem kell külön pontosvesszővel jelezni.

A másik, nem annyira szembetűnőbb, hogy a ”key” változót ott hoztam létre a cikluson belül, mégis látszik azon kívül. Ennek okára szintén később kitérek2.6.

2.5. Értékadás és visszatérési érték

Az értékadásra azért térnék ki részletesebben, mert a Lua támogat egy plusz ”feature”-t a többi nyelvekhez képest. Ez pedig a többszörös értékadás fogalma. Fontos megjegyeznem, hogy ez a fogalom nem idegen a C, C++ és Java nyelveknél sem, de ott a jelentése, hogy ott ugyanazt az

értéket több változónak egyszerre megadhatjuk. Lua-ban viszont kicsit eltér a jelentése. Ugyanis Lua-ben több változónak több különböző értéket adhatunk meg egyszerre.

```
var1, var2, var3 = 1, "a", true
print(var1)  --> 1
print(var2)  --> a
print(var3)  --> true
```

Ha több értéket akarunk megadni a kevesebb változónak, akkor egyszerűen a fel nem használt értékek elvesznek. Nem íródnak be sehova. Ha több a változó, mint az érték, amit értékül adnánk, akkor a változó értéke alapértelmezetten nil lesz.

```
var1, var2 = 1, 1+1, 1+2, 1+3
print(var1)  --> 1
print(var2)  --> 2

var1, var2, var3 = 1, "a"
print(var1)  --> 1
print(var2)  --> a
print(var3)  --> nil
```

És ez hol tud hasznos lenni? Hát például a függvények visszatérési értékeinél. A legtöbb nyelvben egy függvény csak egy dolgot tud visszaadni, legyen primitív típus vagy egy objektum (C#-ban például data objectet is visszaadhatunk, ami adatbázis entitások lekérdezés eredményhalmazát alkothatja, amelyeket dinamikusan is létrehozhatunk, előre nem definiált adatstruktúraként), de attól az még 1 elemnek számít. Lua-ban viszont ez nem kikötés. Felfoghatjuk, hogy értékek listáját adja vissza egy függvény.

Használata rendkívül kényelmes:

```
function foo()
        return 1,2,3
end

var1, var2, var3 = foo()
print(var2, var3) --> 2  3
```

Egy dologgal azonban tisztában kell lenni: hogy az a függvény hány paraméterrel tér vissza. Erre általában a függvény magírója figyelmezteti a használót, így elkerülendő a rossz/nem-kívánt használat.

Viszont emellett vannak további kényelmi funkciói. Például, ha nincs szükségünk minden értékre, amit visszaad egy függvény. Ezeket kétféle módon is megoldhatjuk. Az egyik eset, ha a nemszükséges értékek az utolsók. Ebben az esetben egyszerűen nem veszünk fel annyi változót. Csak a szükségeseket:

```
function foo()
        return 1,2,3
end

var1, var2 = foo()
print(var1, var2) --> 1   2
```

Ebben az esetben a '3' elveszik.

A másik eset, amikor a köztes értékekre nincs szükségünk. Ekkor persze tisztában kell lennünk a visszatérési értékek sorrendjével. De emellett a Lua úgy oldja meg a fölösleges értékek kezelését, hogy bevezet egy ún. **anonymus** változót (dummy varialbe). Ezt "_" szokás jelölni (ilyen jelölést a Kotlin lambda kifejezéseinél szokás használni a nem használt változók jelölésére).

```
function foo()
        return 1,2,3
end

var1, _, var3 = foo()
print(var1, var3) --> 1    3
```

Kiemelném még egyszer, hogy ez rendkívül hasznos tulajdonság mind változó értékadás, mind függvény visszatérési értékével kapcsolatban. Hiszen evvel lehetőség nyílik, hogy egy függvény több dolgot is végezzen egyszerre és a különböző (rész)eredményeket egyszerre adja vissza. Az eredményt pedig aszerint szortírozza a használó, amire éppen szüksége van (nem OO megközelítés, de mivel a Lua multi-paradigm nyelv2.1, ezért van ilyen lehetőség is).

2.5.1. Kiértékelés sorrendje

Muszáj kitérnem erre a részre is egy kicsit. Habár a Lua kiértékelési sorrendje teljesen megegyezik a megszokottakkal (először a jobb oldal értékelődik ki, majd a az értékadás a balnak), az előbbi téma tekintetében enne nagyon fontos jelentés van.

Legszebb példa erre két változó értékének a cseréje.

```
function shuffle()
        size = #deckPool --> deckPool is a table
        for i = size, 1, -1 do
                rand = math.random(size)
                deckPool[i], deckPool[rand] = deckPool[rand], deckPool[i]
        end
end
```

Ezen a példán nagyon szépen látszik a hasznossága a többszörös értékadásnak. Először a jobb oldal kiértékelődik és csak utána történik meg az értékadás. Ezzel lehetőségünk van az értékcserét egy sorban megoldni. Ez a legtöbb nyelvben kn. így nézne ki:

```
a = 1, b = 2
temp = a
a = b
b = temp
-- in lua:
a, b = b, a
```

2.6. Lokális Változók

Az eddigi példákban ügyeltem arra, hogy csak egyfajta változót mutassak be. Tehát az eddigi változók mind globális változóként voltak használva. Ez a fogalom teljesen azonos a C, C++ világában megszokottaktól. De persze ezeket tipikusan szeretik elkerülni csaknem minden program írásakor a programozók, ha lehet. Lua-ban is ez a tipikus helyzet.

Az általános ”konvenció” erre az, hogy csak a könyvtárak és a közös használatra szán modulok legyenek globálisak. Minden más lehetőleg legyen lokális változó. Ennek okai kb. megegyeznek a C, C++ nyelv problémáival: szabadon írhatók, így az egyes objektumok állapotát elronthatják, stb. Ráadásul lua-ban még a típus sem ismert, így lehet, hogy egy teljesen más dolgot ír át az ember, mint amit akar (erre egy megoldás lehet a beszédes változónév, de még ez sem teljes megoldás).

Tehát ha lehet akkor használjunk lokális változókat. De hogy ez mit is jelent? A fogalom teljesen azonos a más nyelvekben használt lokális változó fogalmával (csak egy bizonyos blokkon/környe-

zeten belül definiált és használható változó). Ide legtöbb esetben függvények scope-ját értjük. De Lua-ban értelmezhetünk saját scope-ot is a **do** − **end** szintaxis segítségével, ha szeretnénk.

```
local x = 5
if x == 5 then
        print(x) --> 5
        local x = 1
        print(x) --> 1
end
print(x) --> 5
```

Mivel Lua-ban sorról sorral olvasunk, így nem tudjuk minden esetben a függvény scope-ját kihasználni, hogy felismerje, hogy az ott létrehozott változó legyen lokális vagy sem. Ehelyett a **local** kulcsszóval mondjuk meg Lua-ban, ha egy változót lokálisnak tekintünk.

Jól látszik, hogy az if-en belül definiált 'x' értéke az if scope-ján kívül nem marad meg. Csakúgy, mint más nyelvekben:

```
if expression then
        local x = 1
end
print(x) --> nil
```

Egy másik nagyon fontos megjegyzés a függvények használata. Függvényt kétféle módon tudunk definiálni2.2 és azt is tudjuk, hogy az egyik igazából a másik egy egyszerűbb alakja. Viszont, van egy nagy eltérés, ha lokális függvényt definiálunk:

```
local function foo()
end

local foo = function()
end
```

Az első esetben a függvény neve a függvény scope-jába tartozik, míg a második esetben nem. Ez például egy rekurzív függvénynél rendkívül fontos dolog, amelyre figyelni kell. Hiszen, ha a második módon írjuk meg, akkor hibát fogunk kapni, mert a függvény törzsében a foo-t globális változónak veszi és mivel a jobb oldali érték előbb értékelődik ki, így a foo meghívása még egy olyan változó használatát jelentené, ami eddig lokálisan még nem létezik, így automatikusan globálisnak veszi. Ekkor viszont nem tudja, hogy ez egy függvényt tárol.

2.6.1. Változók listája

Ha már egyszer újra elővettük a változókat, akkor érdemes még egy nagyon fontos dolgot megemlíteni, ez pedig a váltorúk listája (varargs).

```
local function foo(...)
end
        print(...)
end
local a,b,c = 1,2,3
print(foo(a,b,c)) --> 1    2    3
```

Ez a fogalom létezik C++ világában, ott a template meta-programozásnál játszik fontos szerepet, de megjelenik Java-ban is. Ez igazából egy változó, ami mondhatni ismeretlen hosszúságú változók listáját kezeli egy elemként. Ahogy az említett nyelvekben, a Lua-ban is ugyanaz a szerepe. Lokális

változóként van kezelve és a használata nagyban függ attól, hogy a programozó mit is szeretne vele csinálni. Ugyanis, ha csak ezt a paramétert kapja, akkor pontosan tudnia kell, hogy mit takar.

Kicsit hasonló az értékadáshoz2.5. Tekinthetjük úgy, mint egy hosszú listát, amiből szeretnénk lecsípni valamennyit. Erre segítségül a Lua egy ún. **select(i, ...)** függvényt ad. Ez az első 'i' darab változót adja vissza a **'...'**-ból, így már tudjuk, hogy hány változónak fogunk értéket adni. Viszont az még mindig programozási kérdés, hogy mik ezek a változók.

A leggyakoribb használata mégis a függvénydelegálásnál jelenik meg. Hiszen ekkor a meghívott függvénynek a célja nem feltétlenül kapcsolódik a megkapott paraméterekhez, így azokat minden további nélkül továbbadja egy másik függvénynek, aki majd használja őket és tudja is, hogy miket kap.

```
foo = function(table,...)
...
        --> do something
...
        return table:new(...)
end
```

A példán jól láthatjuk, hogy a kapott paraméteren meghívunk egy további függvényt, melynek továbbadjuk a nem használt paramétereket. A kettőspont operátorra majd később kitérünk, egyenlőre fogadjuk el, hogy egy table típusú változónak egy new kulcsú elemét hívtuk meg, mivel az egy függvény.

2.7. Lua standard könyvtárai

Korábban már volt szó arról, hogy mit használunk lokális2.6 és mit globális változóként. Akkor említettem, hogy a kivételes esetek közé tartoznak a Lua beépített könyvtárai. Ezekről említenék meg egy gyors összefoglalást.

Math library

Ez teljesen azonos a más nyelvekben használatos Math könyvárakhoz. Itt nem kell sem #include-olni (mint C++-ban) vagy importálni (mint Java-ban), hanem alapból használhatjuk a **math** globális változón keresztül.

Minden tipikus és gyakran használt függvény definiálva van:

- math.abs
- math.exp
- math.deg
- math.cos
- math.sin
- math.log
- ...

De vannak konstansok is, mint pl. a **math.pi**. De ide tartozik a **math.random(x, y)** függvény is (x és y között generál egy random egészet, ha x = 0 és y = 1, akkor pedig float-ot generál).

String library

A **string** típusú változókat a legkönnyebb és legpraktikusabb a string könyvtárral manipulálni. Ez is teljesen megegyezik pl. a C-ben használt tipikus függvényekkel:

- string.len
- string.lower
- string.upper
- string.match
- string.join
- string.reverse
- string.split
- ...

Egy nagyon gyakran használandó függvény a **string.sub(str, i, j)**. Ez természetesen a substring megfelelője, de azért fontos megemlíteni, mivel a string nem indexelhető, ezért az egyetlen módja, hogy visszakapjuk a string egy karakterét, ha ezt a függvényt alkalmazzuk (ebben az esetben az i = j). Később még lesz szó az index operátorról, amikor is kis csellel és némi következmény árán megoldható egy string indexelése.

Table library

Talán ez egyik legfontosabb könyvár a **table** library, amellyel strukturálisan tudjuk manipulálni a table objektumokat. Mivel a table, mint korábban kiderült2.2.2, egy különös állatfaj, mégis leginkább egy list/map-hez lehet hasonlítani. Más OO nyelvek gyakran definiálnak tömb-manipuláló függvényeket és Lua-ban erre szolgál a table könyvtár:

- table.insert
- table.remove
- table.wipe
- table.sort
- ...

Ez a könyvár elsősorban az 'insert' és 'remove' függvények miatt hasznos. Ezek ugyanis elrejtik előlünk a táblaindexelés aggasztó ismeretét. Tehát nem kell tudni, hogy egy tábla meddig van indexelve és onnan folytassuk tovább a következő index megadásával. Ezek a függvények pont arra jók, hogy ne nekünk kelljen számon tartani az egyes táblák jelenlegi indexeit (ameddig végig iterálhatunk rajtuk index, nem pedig hash szerint). Így utána kényelmesen használhatjuk az **ipairs** függvényt, ami azért hasznos, mert ha egy table-t tényleg listaként szeretnénk értelmezni, a sorrend gyakran fontos lehet.

Debug és System és IO library-k

Pár szó erejéig érdemes megemlíteni ezt a két könyvárat. A **debug** library-t gyakran segítségül hívhatjuk egy problémás kód hibájának megfejtésére (pl. a debug.traceback függvény). Ez hasznos tud lenni, hiszen fordítási nincs lehetőség jelezni a programozónak, viszont ezzel sokat tud segíteni, ha a minimálisnál több adatot ír ki hibaüzenetként.

A másik a **os** könyvtár. Az os library-hoz fordulhatunk például az idő lekérdezése végett, de akár írhatunk eseményeket, amiket szeretnénk, hogy a program kilépése során hívjon meg, mint például a biztos fájl bezárás (**os.exit(files.close())**).

A harmadik és talán leghasznosabb könyvtár az **io** library. E könyvtár segítségével tudunk fájlból olvasni és fájlba írni. Valamint a konzolra tudunk vele írni és olvasni is.

- io.open
- io.close
- io.flush
- io.write
- io.read
- io.lines

A 'write' függvény annyiban tér el a megszokott print-től, hogy a print mindig rak egy sorvége karaktert ha lefutott, míg az io.write nem. Az io.read függvényről még annyit kell tudni, hogy string-ként olvas egy egész sort. Ez majdnem minden más nyelvben így van, ahol dinamikus az olvasás (C-ben például meg kell adni az elvárt bemeneteket típussal pontosítva). A 'lines' függvényt pedig gyakran használjuk fálj sorainak beolvasására. Fontos, hogy nem soronként olvasunk vele. A lines először beolvassa az egész fájlt, és visszaad egy listát, amin végigiterálhatunk.

2.8. A függvények további sajátosságai

2.8.1. Static scoping

A Lua-ban a függvények mindig a saját kontextusukra hivatkoznak. Hogy ez mit is jelent, tekintsük a következő példát:

```
do
        local x = 5
        function foo()
                print(x)
        end
end
x = 1 --> 'x' global variable
foo() --> 5
```

Jól látszik, hogy hiába létezik egy globális változó ugyanavval a névvel, a függvény a saját környezete változóját használja. Azaz sosem arra a környezetre hivatkozik, ahol meg lett hívva.

A függvényeknél általánosságban igaz, hogy mindig "belülről kifele" halad. Azaz először a saját környezetében keresi a változókat, amiket használ és csak utána a globálisakat. Más nyelvekben, mint például C++ vagy Koltin, is hasonló működést tapasztalhatunk.

2.8.2. Closure

Még mielőtt belekezdenénk ebbe a témába, muszáj kitérnem, még egyszer a függvények visszatérési értékére. Az eddig bemutatottak mellett kiegészítem avval a tulajdonsággal, hogy függvény képes visszaadni egy másik függvényt is. Elsőre ez nem meglepő, hiszen a 'function' is egy típus és eddigi tudásunk alapján a függvény bármilyen típus(oka)t visszaadhat.

Nézzük a következő példát:

```
function counter()
        local x = 0
        return function()
                x = x + 1
                print(x)
        end
end
```

Itt a függvényünk létrehoz egy lokális változót majd visszatér egy függvénnyel. Mint azt az előző témában is bemutattuk, egy függvény a saját kontextusát használja ki.

```
c1 = counter()
print(c1()) --> 1
print(c1()) --> 2
```

Ezen a példán tejesen jól látszik a static scoping jelentősége.

```
c2 = counter()
print(c2 == c1) --> false
```

Érdekes megfigyelni, hogy a két változó értéke nem ugyanaz. De vajon miért? Hiszen ugyanazt a függvényt kapták vissza nem? Igazából nem. A **function() ... end** meghívása után egy új függvény jön létre mindig. Fontos megjegyezni, hogy ez nem ugyanaz az eset, mint a következő:

```
c3 = counter
```

Ekkor ugyanis nem fut le a function, csak a referencia másolódik.

Tehát a függvény meghívása egy teljesen új függvényt hoz létre. És vajon a környezete is új?

```
print(c2()) --> 1
print(c1()) --> 3
```

Ezen a példán jól látszik, hogy a számláló is különbözik a két függvénynél. Tehát nem csak egy új függvényt, hanem egy új lokális változót is létrehozott a Lua. Mind a két függvény különböző környezetet lát. Azokat a függvényeket, amik megőrzik a kontextusukat, **closure**-nek hívjuk. És azok a változók, amik "megőrződnek" a closure-ök által az ún. **upvalues** a Lua-ban. Mivel igazából a **function() ... end** meghívása után mindig egy új closure keletkezik, ezért valójában az összes függvény a lua-ban closure-ként viselkedik.

Térjünk újra vissza arra, hogy egy függvény akár több dolgot is visszaadhat (akár több függvényt is), vagy akár egy table típusú objektumot, aminek az elemei a függvényünkben létrehozott újabb függvények. Ezek mind ugyanazt a kontextust látják, azaz ugyanahhoz a scope-hoz vannak kötve. Erre a tulajdonságra később visszatérünk és akkor bemutatom mekkora szerepe van ennek például az objektum-orientáltságban.

2.8.3. Hooking

A függvények egy további hasznos tulajdonsága a hooking. Nyersen lefordítva nagyjából azt jelenti, hogy "beleakaszkodik". Ahhoz, hogy eszt megértsük egy egyszerű példán bemutatom a működését és így egyből értelmet nyer a jelentés:

```
do

local old = print
print = function(...) --> redefine print command
        return old(os.date("%H:%M:%S", os.time()), ...) --> new print
end

end

print("Hi") --> 12:54:15 Hi
```

Tehát a hookingot arra használhatjuk, hogy meglévő függvényeink feladatát kiegészíthessük, plusz funkciókkal bővítsük. Ha jól megfigyeljük, ezzel akár a 'function overload'-ot is megvalósíthatjuk.

És az a szép a Lua-ban, hogy ezt bárhol megírhatjuk, ahol épp szükségünk van rá. Csak elmentjük egy lokális változóban az eddig használt függvényt, újradefiniáljuk a hooking segítségével és ha

már nem használjuk, akkor visszaállítjuk a régit. Vagy, ha csak egy helyen akarjuk kihasználni, akkor az még egyszerűbb, hiszen a lokális környezett miatt máshol úgy sem látja a Lua ezt a "kiegészítést", így ott az eredetit használja. Viszont fontos, hogy tényleg lokális környezetben (akár closure-ban) legyenek és nem globálisan írjuk meg. Ezt a módszert akár wrapper függvényként is értelmezhetjük, de inkább egy függvény viselkedésének kiegészítésére használjuk.

2.9. Meta-táblák és meta-metódusok

Lua-ban minden értéknek van egy úgynevezett **metatable**-je. A metatable egy tipikus table objektum, ami képes különböző dolgok tárolására. A metatable-ket arra használják, hogy az egyes értékek vielkedését módosítsák. HOgy ez mit is jelent? Minden típusnak megvannak a maga standard operátorai és, hogy ehhez milyen függvények tartoznak. Ezeket az információkat tartalmazzák a metatable-ök.

Magán a table típuson kívül viszont minden más típus metatable-je megegyezik. Ez annyit jelent, hogy ha egy bizonyos string típusú változó összeadó operátorát akarná az ember átírni, akkor összes string operátorát megváltoztatja. A table típusú változók esetében minden table-nek saját metatable-je van (emlékeztető: a t = , leírása mindig új table-t hoz létre, egyúttal új metatable-t is).

A Lua beépített függvényekkel teszi lehetővé a metatable-ök elérését: **setmetatable(table, metatable), és getmetatable(table)**. A függvények kinézete a C nyelvre emlékeztet leginkább, ahol még nem voltak önálló objektumok saját viselkedéssel (függvényekkel). A 'setmetatable' segítségével az első paraméterként megadott table metatable-jét állítjuk be a második paraméterként megadott table-lel. Sajnos jól látszik, hogy ilyenkor az egész metatable-t felülírja (kicseréli). Tehát érdemes odafigyelni, hogy nem lehet sorba fűzni újabb és újabb operátor overload-okat ezzel a módszerrel.

```
local Player = {}

setmetatable(Player, {
                __call = function(class,...)
                        local newPlayer = class:new(...)
                        newPlayer.currentDeckIndex = 1
                        newPlayer.numberOfUsedDecks = 1
                        newPlayer.decks = class:createDecks(newPlayer)

                        return newPlayer
                end
})
setmetatable(Player, {
                __index = Participant
})

print( Player() ) --> error, attempt to call a table value (upvalue 'Player')
```

```
local Player = {}

setmetatable(Player, {
        __call = function(class,...)
                        local newPlayer = class:new(...)
                        newPlayer.currentDeckIndex = 1
                        newPlayer.numberOfUsedDecks = 1
                        newPlayer.decks = class:createDecks(newPlayer)

                        return newPlayer
                end,
        __index = Participant
})
```

Jól látható a probléma az első esetben, hiszen a **__call** operátor nem volt inicializálva a második setmetatable után, így a függvényhívás hatására nem volt semmi sem beállítva, ami hibát eredményezett.

2.9.1. Alapértelmezett meta-metódusok

Mint ahogy az előbbiekben elmagyaráztam a metatable-ben vannak az ún. meta-metódusok (továbbiakban metamethod). A metatable-ben speciális elemek vannak, amelyek tárolják a megszokott operátorok alapvető viselkedéseit. Ezeket a viselkedéseket függvények formájában értjük és őket hívjuk **metamethods**-nak.

A speciális kulcsokat a table-ben két aláhúzás karakterrel kezdjük és aztán jön a neve, amely utal arra, hogy milyen alapvető viselkedést definiál. A Lua a következő alap metamethod-okat szolgáltatja (ezek a metódusok két paramétert várnak):

- _add: + operátor
- _concat: a .. operátor (összefűzés)
- _div: / operátor
- _eq: == operátor
- _lt: < operátor
- _le: <= operátor
- _mul: * operátor
- _mod: % operátor
- _pow: ^ operátor
- _sub: – operátor
- _unm: negálás operátor
- _add: + operátor
- ...

Ezeken kívül van még 4 különleges operátor, amiket érdemes külön-külön kicsit jobban megismerni:

- _len: a **#** operátor. Ahogy a string-nél is megmutattam ez a hossz operátor, ami a string-ek esetében a szöveg hosszát adja meg (valószínűleg a string-len függvényt használva). Viszont gyakran szokás table típusú változóknál is használni, ugyanis alapértelmezetten visszaadja egy table-ben tárolt kulcs-érték párok számát. Azonban van egy fontos szabály: table objektum metatable-jére nem használható, mivel metatable-öket csak table-ökhöz csatolva tudunk használni.
- _call: a függvényhívás operátor. Fontos tulajdonság, hogy ez a függvény első paramétereként a table-t kapja és utána a többit, amit a függvénynek átadtak (a fenti példában 2.9 ez szemléletesen látszik).

- _newindex: ez a metódus akkor hívódik meg, ha megpróbálunk értékül adni egy új értéket egy olyan mezőnek, ami nincs a table-ben. Ez persze nem hívódik meg újra, ha már létezik az a "kulcs". Fontos megjegyzés, hogy az új mező felvétele megtörténik mindenképp, ha eddig nem létezett. Ez a függvény mint egy event-ként viselkedik (hasonlót C#-ban láthattunk).
- _index: ez a metódus akkor hívódik meg, ha hozzá akarunk férni a table egy mezőjéhez, ami még nem létezik. Fontos különbség a '_newindex'-hez képest, hogy itt nem történik értékadás, azaz nem is hoz létre új kulcs-érték párt. Itt megpróbálja elérni a "kulcs" által meghatározott értéket, és ha nincs ilyen kulcs a table-ben, akkor hívódik meg.

2.10. Objektum-orientáltság

Mint tudjuk, a Lua-ban megvalósítható az objektum-orientált programozás. A Lua támogatja az ún. prototipikus objektum-orientált programozást. Ezt pedig az _index metametódus segítségével éri el.

Nézzük, hogy mit is értünk prototipikus objektum-orientáltságon. Először is definiálunk egy table (továbbiakban táblaként is hívom), amiben függvényeket és különböző értékeket (attribútumokat) fogunk tárolni. Ez a táblát tekintjük az ún. **prototípus**nak.

2.10.1. Kettőspont operátor

Még mielőtt belekezdenénk, hogy hogyan is lehet megvalósítani az OO stílust Lua-ban, először térjünk ki egy fontos dologra a táblákkal kapcsolatban. Ez pedig a kettőspont operátor. Az eddigiekben, ha szerettünk volna egy függvényt felvenni a táblába, akkor egy változóban eltároltuk és így hivatkoztunk rá:

```
table = {}
table.foo = function()
        print(Hello)
end

table.foo() --> Hello
```

Mint egy tábla adattagja. Viszont van egy másik módszer, hogy definiálni tudjunk egy függvényt egy tábla attribútumaként:

```
Table = {
        tableView
}
function Table:noticeUpdate()
        self.tableView:update()
end

Table:noticeUpdate()
```

Itt láthatjuk, hogy a másik módszerben már a kettőspont operátort használjuk. Ez egy plusz tulajdonsággal ruházza fel a függvényt. Ebben az esetben, amikor meghívjuk a függvényt, impliciten megkapja paraméterként a táblát, mint paraméter. És a **self** kulcsszó segítségével tudjuk felhasználni.

Jól érezhető, hogy ez mennyire hasonlít az OO nyelvekben megszokott **this** kulcsszóra. Igazából lényegében azzal egyenértékű, hiszen így a táblát elérve hozzáférünk minden más függvényhez, illetve attribútumhoz. A példában pl. így el tudjuk érni a 'tableView' adattagot és az, hogy rajta is meghívunk egy függvényt a kettőspont operátorral, már következtethetünk arra, hogy az is egy tábla.

Talán már lehet érezni, hogy ez mennyire segít majd nekünk az OO-s szemlélet megvalósításában. Habár érdemes fejben tartani, hogy ez is csak egy egyszerűbb írásmód, semmi más. Valójában ennek a megfelelője:

```
function Table.noticeUpdate(self)
        self.tableView:update()
end

Table.noticeUpdate(Table)
```

Itt talán érdemes megjegyezni, hogy az ilyen ”megoldások” mennyire hasonlítanak a C nyelvben megszokott ”adjuk át paraméterként a . . . ’ függvénymegadásokhoz. Ez nem véletlen. A Lua ugyanis nagyban kötődik a C nyelvcsaládhoz. Megfelelő libraryk segítségével akár a lua kódunkat C kódra is át tudjuk írni és ezek a megoldások a könnyű olvasást és ”átírást” teszik lehetővé.

2.10.2. Konstruktor

Azt már a függvények visszatérési értékeinél 2.5 is megtudtuk, hogy egy függvény visszatérhet több dologgal is, gyakran szeretjük ezeket egy közös halmazban tenni ha az adatok úgyis összetartoznak. Erre kényelmesen használhatjuk a table típust. Így ezzel nagyjából már el is tudjuk képzelni, hogy hogyan is néz ki egy konstruktorhívás. Kicsit pontosabban a C nyelvben megszokott módszer rövid összefoglalója volt ez. Lényegében bármilyen objektumot létre tudunk hozni és a függvény visszatérési értékét elmentjük egy változóba.

Azonban mi nem szeretnénk egy ’createXXX()’ függvény írni, hanem a C++ és Kotlin nyelvekben megszokott módon szeretnénk létrehozni az új objektumokat:

```
cell = CreateCell() --> C style constructor

cell = Cell() --> C++, Kotlin, ... style constructor
```

A következőkben ezt az elképzelést valósítjuk meg.

2.10.3. Prototípus és objektum

Először is meg kell tervezni a prototípust, amit majd felhasználunk az objektumok létrehozásához.

```
local Cell = {
        state = "",
        changeState = 0
}

function Cell:die()
        if(self.changeState == ChangeType.DIE) then
                self.state = StateType.DEAD
        end
end

function Cell:birth()
        if(self.changeState == ChangeType.BIRTH) then
                self.state = StateType.ALIVE
        end
end

...
```

Most pedig a következő lépés: megcsinálni a konstruktorunkat:

```
function CreateCell(state)
        local newCell = {}
        newCell.state = state
        newCell.neighbors = {}
        return newCell
end
```

Ez így viszont még nincs teljesen kész. Hiszen ez még csak annyit csinál, hogy létrehoz egy táblát, hozzáad pár attribútumot és visszaadja azt. A prototípust még valahogy hozzá kell kötni ehhez az új táblához. Nézzük a következő kódot:

```
local metatable = {
        __index = Cell
}

function CreateCell(state)
local newCell = setmetatable({}, metatable)
newCell.state = state
newCell.neighbors = {}
return newCell
end
```

Itt ami elsőre szembetűnik, hogy az _index metamethod nem függvényt kapott hanem egy táblát (jelen esetünkben a prototípust). Ezt azért tehetjük meg, mert a Lua ad erre támogatást (a függvényhívás teljesítmény visszaeséssel járhat így ez egy egyszerűbb megoldás). De vajon mit is jelent ez valójában:

```
local metatable = {
__index = function(t, k)
                return Cell[k]
        end
}
```

Csupán ennek a függvénynek a rövidítését. Ahol a 't' az az a tábla, amin az _index metódus meghívódott és a 'k' pedig a kulcs, amivel meghívták.

így már nagyon egyszerű értelmezni ezt:

```
local metatable = {
        __index = Cell
}
```

Tehát, ha egy tábla egy nemlétező attribútumát (ami alatt függvényt tároló változót is érthetünk) hívjuk meg, akkor az _index metódus által megnézi a prototípusunkban és ha talál, akkor meghívja azt. Ezzel a példánkban elérhetővé váltak a prototípus függvényei és előre definiált attribútumai.

Most már csak meg kéne valósítani, hogy úgy viselkedjen, mint egy tipikus konstruktor. Erre felhasználhatjuk a _call metamethod-ot. Azaz beállítjuk, hogy ha valaki meghívja a táblán a () operátort, akkor hívja meg a konstruktorunkat. Ezzel elrejtve azt.

```
local Cell = {
        state = "",
        changeState = 0
}

setmetatable(Cell, {
        __call = function(class, ...)
                return CreateCell(...)
        end
})

local metatable = {
__index = Cell
}

function CreateCell(state)
        local newCell = setmetatable({}, metatable)
        newCell.state = state
        newCell.neighbors = {}
        return newCell
end

local cell = Cell()
```

Ezen jól látható, hogy a prototípusnak beállítottuk, hogy ha függvényhívás operátort használják, akkor hozzon létre egy új táblát és használja a prototípus függvényeit.

Ezzel nagyjából készen is lennénk, de elég sok plusz kódot kéne írni. Nem mellesleg maga a konstruktor nem tartozik magához a prototípushoz. És azt sem használtuk ki, hogy valójában a _call hívásánál mi átadtuk magát a prototípust. Kicsit alakítva a kódot a következő, komplexebb, de mindenképp szebb kódot kapjuk:

```
local Cell = {
        state = "",
        changeState = 0
}

setmetatable(Cell, {
        __call = function(class, ...)
                return class:CreateCell(...)
        end
})

function Cell:CreateCell(state)
        local newCell = setmetatable({}, self)
        self.__index = self
        newCell.state = state
        newCell.neighbors = {}
        return newCell
end

local cell = Cell()
```

Vizsgáljuk meg, hogy mi is történt. Először is kihasználtuk, hogy a _call metódus megkapja a prototípust. Ezt fel tudjuk használni, hogy a konstruktorunkat a prototípushoz kössük, mint függvényt, hiszen meg tudjuk hívni. Ráadásul tovább is tudjuk adni a kettőspont operátor segítségével. Így a konstruktor már rendelkezik a prototípussal és nem kell egy globális változót csinálnunk, hogy elérjük azt. Ha viszont már megvan a prototípus, nem kell mást csinálni, csak beállítani az új objektum metatable-jeként. Így a prototípus metatable-je megegyezik az új objektum metatable-jével.

A következő sor elsőre furán olvasható, de ha egyszerűen odaképzeljük a self helyett a Cell-t, akkor máris megértjük, hogy a 'metatable' létrehozását oldjuk meg egy sorban. Ekkor persze a prototípus _index-ét is beállítjuk magára a prototípusra, de ez teljesen jó (hiszen így érjük el, hogy az új objektum elérhesse a prototípust). Nem csak az új objektum, de a prototípus is saját magát látja és az ő attribútumai között keres. Van még egy további haszna is, de ezt majd az öröklésnél kifejtem.

Végeredményként teljesen ugyanazt a kódok kapjuk, de mégis szebb és kompaktabb a második, hiszen kihasználunk minden lehetőséget amit a metametódusok és a kettőspont operátor biztosítanak.

Még egy fontos dolog, amire mindenképp oda kell figyelni, amikor a konstruktorokat írja az ember. Először is: a prototípus inkább a közös viselkedésre való. Azaz a függvények legyenek elsősorban a prototípusban. Az egyéb változókat, mivel úgyis új objektum jön létre, nyugodtan létrehozhatjuk ott is, nem kell feltétlen a prototípusban felvenni őket. Sőt, a table-öket kifejezetten tilos. Ugyanis, ilyenkor minden új objektum ugyanazt a table-t látja és írja/olvassa (úgy viselkedik, mintha static addatagja lenne egy osztálynak pl. C++-ban). Ezt inkább el szeretnénk kerülni. Így a megoldás, hogy ezeket mindig a konstruktorban, az új objektumnak hozzuk létre.

2.10.4. Öröklés

Az utolsó dolog az OO világában, amire ki kell térnem az az öröklődés. A Lua erre szintén biztosít lehetőséget, méghozzá egy nagyon egyszerű módon. Csupán az előbbi gondolatmenetet kell kiegészíteni.

Vegyünk fel egy ősosztályt:

```
local Participant = {}

setmetatable(Participant, {
        __call = function(class,...)
                return class:new(...)
        end
})

function Participant:new(amountOfMoney, dealer, behavior, inputHandler, observer)
        local newParticipnat = setmetatable({}, self)
        self.__index = self
        newParticipnat.amountOfMoney = amountOfMoney or 100
        newParticipnat.behavior = behavior
        newParticipnat.dealer = dealer
        newParticipnat.inputHandler = inputHandler
        newParticipnat.observer = observer
        return newParticipnat
end
```

Ez teljesen hasonló módon tesszük meg, mint az előző példa esetén. Ezután nézzük meg, hogy hogyan néz ki egy leszármazottja:

```
local Player = {}

setmetatable(Player, {
        __call = function(class,...)
        local newPlayer = class:new(...)
        newPlayer.currentDeckIndex = 1
        newPlayer.numberOfUsedDecks = 1
        return newPlayer
        end,

        __index = Participant
})

player = Player()
print(player.amountOfMoney) --> 100
```

Nézzük meg mi is történt itt valójában, illetve mik is a változások az előző kódhoz képest. Az első szembetűnő dolog, hogy amikor a _call metódust felülírjuk, akkor látjuk, hogy a meghívjuk a **new** függvényt, holott a Player prototípus-nak nincs is ilyen függvénye. Ezt azért teheti meg, mert az _index-et is felüldefiniáltuk, mégpedig az ős prototípusával. Így már meg tudja hívni a 'new' függvényt.

Viszont érdemes megfigyelni, hogy a 'new' meghívásánál a 'class' igazából maga a Player prototípus. Tehát a 'self' paraméter már nem a 'Participant' tábla lesz, hanem a Player. Beállítjuk , hogy az új játékos metatable-je legyen a Player prototípus. Utána pedig az _index-et is beállítjuk, hogy a Player prototípusra mutasson. Ugye ekkor felmerül a kérdés, hogy ez mind szép és jó, de hogyan tudom akkor használni a Participant adattagjait? Egyszerűen úgy, hogy a Player _index-e pedig a Participant prototípusra mutat. Ez ezt jelenti, hogy először a Player táblában keresi és, ha ott nincs, akkor megnézi a Participant táblában. Ez a példa pedig szépen bemutatja, hogy a prototipikus öröklés hogyan valósul meg.

Pár fontos észrevétel azonban elengedhetetlen. Az első és szembetűnőbb, hogy a Player objektum mennyire értelmezhető Participant-nak is egyszerre. Igazából egy új Participant objektum nem jön létre, csak egy Player, ami tudja használni az őse függvényeit/attribútumait. De a Participant _call metódusa meg sem hívódott. Más OO nyelvekben az ős konstruktora is lefut (sőt tipikusan meg kell hívni azt). Itt nem.

Ez a gondolatmenet továbbvíve felveti az absztrakció és az interfész hiányának problémáját egyaránt.

Lua-ban nem tudunk ilyen dolgokat definiálni. Egy későbbi fejezetben megmutatok egy lehetséges ”megoldást’, amellyel interfész szerű viselkedést el tudunk érni.

A másik fontos dolog pedig, hogy a Player prototípus metatable-jének beállítása során egyszerre két metódust is definiálni kell jelen esetben. Hiszen egy korábbi példán 2.9 bemutattam, hogy milyen következményei vannak, ha külön-külön próbálnánk beállítani őket.

Zárásul még egy dologgal szeretném kiegészíteni az öröklést, mégpedig, hogy a Lua-ban lehetőség van a többszörös öröklésre (ezt csak nagyon kevés nyelv tudja elmondani magáról pl. C++). És annak ellenére, hogy ahol ezt egyen talán meg lehet valósítani, ott rendkívül sok odafigyelése és szabályra kell odafigyelni, addig Lua-ban ez rendkívül egyszerű módon megoldható.

```
local Driver = {}

setmetatable(Driver, {
        __index = Driver
})

function Driver:canDrive(age)
        if(age > 17) then
                return true
        else
                return false
        end
end

local Player = {}

setmetatable(Player, {
__call = function(class,...)
local newPlayer = class:new(...)
newPlayer.currentDeckIndex = 1
newPlayer.numberOfUsedDecks = 1
return newPlayer
end,

__index = function(t, k)
                return Participant[k] or Driver[k]
        end
})

player = Player()
print(player.amountOfMoney) --> 100
print(player:canDrive(16)) --> false
```

Látható, hogy csak az _index metódus változott meg. Mivel nem egyértelmű, hogy melyik prototípust használjuk, így a rövidítés nem lehetséges, tehát egy függvényt kell megvalósítani. Ez viszont egy elég egyszerű függvény és a megértése szinte evidens. Ha nem találja a meghívott attribútumot a Participant-ban, akkor megnézi a Driver-ben. Ilyen egyszerű az egész.

Persze felmerülhet olyan kérdés, hogy a logikai vesztegzár miatt, ha mind a kettő rendelkezik egy ugyanolyan nevű függvénnyel, akkor sosem érjük el a második függvényét. De ekkor inkább az a kérdés merül fel, hogy ha mind a kettő használ azonos függvényt, akkor vajon jó-e maga a modell. Ez alatt azt értem, hogy lehetséges, hogy a Participant és a Driver egy közös ősből származik és a kérdéses függvényt az öröklési szinten feljebb kellene tolni (tipikus OO szemlélet). A többszörös öröklődés lehetősége viszont nem szokott ösztönző példa lenni. Csak jó tudni, hogy ez is megvalósítható, méghozzá nagyon egyszerű módon.

3. fejezet

A Kotlin Nyelv és sajátosságai