Témalabor dokumentáció
Változatos programozási nyelvek és környezetek megismerése

Szalai Dávid
AGGRW2

konzulens:
Kövesdán Gábor

2018. december 2.

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.

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 az 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.

3. fejezet

A Kotlin Nyelv és sajátosságai