**UNIVERZITA KONŠTANTÍNA FILOZOFA V NITRE**

FAKULTA PRÍRODNÝCH VIED

V2V komunikácia použitím minipočítača Raspberry Pi

DIPLOMOVÁ práca

**Nitra 2020 DÁVID BOJNANSKÝ**

**UNIVERZITA KONŠTANTÍNA FILOZOFA V NITRE**

FAKULTA PRÍRODNÝCH VIED

V2V KOMUNIKÁCIA POUŽITÍM MINIPOČÍTAČA RASPBERRY PI

DIPLOMOVÁ práca

Študijný odbor: 9.2.9 Aplikovaná informatika

Študijný program: Aplikovaná informatika

Školiace pracovisko: Katedra informatiky

Školiteľ: PaedDr. Peter Švec, PhD.

Nitra 2020 Dávid Bojnanský



# Poďakovanie

Touto cestou vyslovujem poďakovanie pánovi PaedDr. Petrovi Švecovi, PhD. za odporúčania, rady a pripomienky počas písania tejto práce, ako aj nasmerovania a uvedenia do problematiky na začiatku.

# Abstrakt

BOJNANSKÝ, Dávid: V2V komunikácia použitím minipočítača Raspberry Pi. [Diplomová práca]. Univerzita Konštantína Filozofa v Nitre. Fakulta prírodných vied. Školiteľ: PaedDr. Peter Švec, PhD. Stupeň odbornej kvalifikácie: Magister odboru Aplikovaná informatika. Nitra: FPV, 2020. 51 s.

Diplomová práca sa venuje problematike programovacieho jazyka Lua a tvorbe vzorových aplikácií. Bakalárska práca je rozdelená na teoretickú a praktickú časť. Teoretická časť je zameraná na analýzu súčasného stavu a je rozdelená na 3 časti. Prvá časť sa zaoberá programovacími paradigmami, ktoré programovací jazyk Lua využíva. Druhá časť sa zaoberá charakteristikou programovacieho jazyka Lua. Tretia časť poskytuje praktický pohľad na programovací jazyk Lua. Praktická časť je zameraná na analýzu vlastností, konceptov a postupov typických pre programovanie v programovacom jazyku Lua. Túto analýzu dopĺňajú vzorové aplikácie, ktoré názorne demonštrujú programovací jazyk Lua v praxi. Vo vzorových aplikáciách je dôraz kladený hlavne na ich komplexnosť a jednoduchosť. Bakalárska práca sa tiež zaoberá vložením do hostiteľskej aplikácie, čo názorne demonštrujeme na programovacom jazyku PHP a obchodnej platforme *FXCM Trading Station*.

Kľúčové slová: V2V. Raspberry Pi. OBD-II.

# Abstract

BOJNANSKÝ, Dávid: V2V communication using minicomputer Raspberry Pi. [Diploma Thesis]. Constantine the Philosopher University in Nitra. Faculty of Natural Sciences. Supervisor: PaedDr. Peter Švec, PhD. Degree of Qualification: Master of Applied Informatics. Nitra: FNS, 2020. 51 p.

Diploma thesis deals with Lua programming language and creation of sample applications. Bachelor thesis is divided into theoretical and practical part. The theoretical part is focused on analyzing of the current situation and is divided into 3 parts. The first part deals with programming paradigms used by Lua programming language. The second part deals with the characteristics of the programming language Lua. The third part provides a practical insight into the Lua programming language. The practical part is focused on analysis of the characteristics, concepts and procedures specific for the programming in the programming language Lua. This analysis is complemented by the sample applications that demonstrates the Lua programming language in practice. In the sample applications the focus is mainly on their complexity and simplicity. Bachelor thesis also deals with the embedding into the host application, which we demonstrate on the PHP programming language and trading platform *FXCM Trading Station*.

Keywords: V2V. Raspberry Pi. OBD-II.

# Obsah

[Úvod 9](#_Toc448063776)

[1 Analýza súčasného stavu programovacieho jazyka Lua 10](#_Toc448063777)

[1.1 Lua a Programovacie paradigmy 10](#_Toc448063778)

[1.1.1 Imperatívna paradigma 11](#_Toc448063779)

[1.1.2 Procedurálna paradigma 11](#_Toc448063780)

[1.1.3 Funkcionálna paradigma 11](#_Toc448063781)

[1.1.4 Objektovo orientovaná paradigma 12](#_Toc448063782)

[1.2 Charakteristika programovacieho jazyka Lua 13](#_Toc448063783)

[1.2.1 Rozšíriteľnosť 14](#_Toc448063784)

[1.2.2 Jednoduchosť a veľkosť 14](#_Toc448063785)

[1.2.3 Rýchlosť 14](#_Toc448063786)

[1.2.4 Prenositeľnosť 15](#_Toc448063787)

[1.2.5 Vložiteľnosť 15](#_Toc448063788)

[1.3 Lua v praxi 16](#_Toc448063789)

[1.3.1 Aplikácie 16](#_Toc448063790)

[1.3.2 Moduly 17](#_Toc448063791)

[1.3.3 Online zdroje 17](#_Toc448063792)

[1.3.4 Odborná literatúra 18](#_Toc448063793)

[1.3.5 Vývojové prostredia 18](#_Toc448063794)

[2 Ciele bakalárskej práce 19](#_Toc448063795)

[3 Analýza programovacieho jazyka Lua 20](#_Toc448063796)

[3.1 Základy 20](#_Toc448063797)

[3.1.1 Typy hodnôt 21](#_Toc448063798)

[3.1.2 Operátory 22](#_Toc448063799)

[3.1.3 Riadiace konštrukcie 23](#_Toc448063800)

[3.2 Dátové štruktúry 24](#_Toc448063801)

[3.2.1 Tabuľky 24](#_Toc448063802)

[3.2.2 Polia 25](#_Toc448063803)

[3.2.3 Mapy 25](#_Toc448063804)

[3.2.4 Sety 25](#_Toc448063805)

[3.2.5 Dvojkoncové fronty 26](#_Toc448063806)

[3.2.6 Spájané zoznamy 26](#_Toc448063807)

[Vzorový príklad pr3\_2 26](#_Toc448063808)

[3.3 Funkcie 27](#_Toc448063809)

[3.3.1 Definovanie funkcie 27](#_Toc448063810)

[3.3.2 Volanie funkcie 28](#_Toc448063811)

[3.3.3 Pomenované argumenty 28](#_Toc448063812)

[3.3.4 Variabilný počet argumentov 28](#_Toc448063813)

[3.3.5 Viacnásobný výsledok 29](#_Toc448063814)

[3.3.6 Anonymné funkcie 30](#_Toc448063815)

[Vzorový príklad pr3\_3 30](#_Toc448063816)

[3.4 Korutiny 30](#_Toc448063817)

[Vzorový príklad pr3\_4 32](#_Toc448063818)

[3.5 Chyby 32](#_Toc448063819)

[3.5.1 Ošetrenie chýb 34](#_Toc448063820)

[Vzorový príklad pr3\_5 34](#_Toc448063821)

[3.6 Metatabuľky a metametódy 34](#_Toc448063822)

[3.6.1 Aritmetické metametódy 36](#_Toc448063823)

[3.6.2 Bitové metametódy 36](#_Toc448063824)

[3.6.3 Porovnávacie metametódy 37](#_Toc448063825)

[3.6.4 Tabuľkové metametódy 37](#_Toc448063826)

[3.6.5 Ostatné metametódy 38](#_Toc448063827)

[Vzorový príklad pr3\_6 39](#_Toc448063828)

[3.7 Objektovo orientované programovanie 39](#_Toc448063829)

[3.7.1 Návrh triedy 39](#_Toc448063830)

[3.7.2 Návrh odvodenej triedy 41](#_Toc448063831)

[3.7.3 Návrh triedy so súkromnými atribútmi 42](#_Toc448063832)

[Vzorový príklad pr3\_7 42](#_Toc448063833)

[3.8 Moduly 43](#_Toc448063834)

[3.8.1 Submoduly 44](#_Toc448063835)

[3.8.2 Načítanie modulu 44](#_Toc448063836)

[Vzorový príklad pr3\_8 45](#_Toc448063837)

[3.9 Vloženie do hostiteľskej aplikácie 45](#_Toc448063838)

[3.9.1 Vloženie cez programovací jazyk C 46](#_Toc448063839)

[3.9.2 Vloženie cez programovací jazyk Java 46](#_Toc448063840)

[3.9.3 Vloženie cez programovací jazyk PHP 46](#_Toc448063841)

[Vzorový príklad pr3\_9/php: PHP aplikácia volajúca Lua skript 47](#_Toc448063842)

[Vzorový príklad pr3\_9/fxcm: Indikátor predaja a nákupu 47](#_Toc448063843)

[Záver 48](#_Toc448063844)

[Zoznam bibliografických odkazov 49](#_Toc448063845)

[Zoznam príloh 51](#_Toc448063846)

# Zoznam ilustrácií a tabuliek

Obrázok 1 Blížiace sa policajné alebo záchranné vozidlo 11

Obrázok 2 Blížiace sa policajné alebo záchranné vozidlo 12

Graf 3 Porovnanie časov potrebných na vyriešenie sudoku v rôznych programovacích jazykoch 15

# Úvod

Programovací jazyk Lua je v súčasnosti celosvetovo rozšírený a uznávaný. Veľké uplatnenie si našiel predovšetkým v hernom priemysle. Väčšinu učebných materiálov nájdeme v anglickom jazyku, čo pre niektorých ľudí môže ešte stále predstavovať prekážku. Výber bakalárskej práce bol založený na snahe oboznámiť sa, pre nás dovtedy s neznámym programovacím jazykom, a tak rozšíriť naše vedomosti o jeden ďalší.

Cieľom diplomovej práce je preskúmať a popísať vlastnosti, koncepty a postupy zaužívané v programovacom jazyku Lua. Druhým cieľom je, na základe získaných vedomostí, vytvoriť kolekciu vzorových aplikácií, ktoré tieto vlastnosti, koncepty a postupy demonštrujú v praxi.

V diplomovej práci sa opierame hlavne o informácie z knihy *Programming in Lua*, ktorej autorom je samotný vynálezca programovacieho jazyka Lua, a oficiálnu webovú stránku *www.lua.org*.

V prvej kapitole sa zaoberáme analýzou súčasného stavu V2V komunikácie. V rámci tejto kapitoly rozoberáme nedávnu históriu z roku 2007, kde boli prvé náznaky na takúto komunikáciu, ...

V druhej kapitole definujeme ciele, ktoré realizujeme v rámci tretej kapitoly.

V tretej kapitole sa venujeme jednotlivým aspektom programovacieho jazyka Lua. V rámci nej rozoberáme najskôr základné aspekty jazyka ako úvod do neho, dátové štruktúry a funkcie. Následne sa presúvame k pokročilejším aspektom jazyka ako korutiny, chyby, metatabuľky a metametódy, objektovo orientované programovanie a moduly. Na záver sa zaoberáme možnosťami jeho vloženia do hostiteľskej aplikácie. Aby sme jednotlivé aspekty demonštrovali v praxi, vytvorili sme kolekciu vzorových aplikácií.

# 1 Analýza súčasného stavu V2V komunikácie

Prvé náznaky a ambície na komunikáciu V2V boli už okolo roku 2007. Označenie V2V predstavuje skratku pre anglický výraz *Vehicle-to-Vehicle*, teda po našom *Vozidlo-Vozidlo*. Keď k tejto skratke pridáme slovo *komunikácia*, tak spolu to predstavuje komunikáciu viacerých vozidiel medzi sebou.

## 1.1 Prvé náznaky už v roku 2007

Podľa článku publikovaného na webstránke denníka SME, v sekcii o autách, dôvodom na vôbec začatie rozmýšľania o komunikácii medzi vozidlami bola nemecká štatistika o nehodovosti na cestách z roku 2003. Tá hovorila jasnými číslami, a teda že nehodovosť má rastúcu tendenciu. Autor v článku uvádza: „26,1 % havárií bolo spôsobených jazdou neprimeranou rýchlosťou a nedodržiavaním bezpečných odstupov medzi vozidlami, 22,8 % príčin tvorili prekážky na ceste, v zákrute, kľučkovania medzi autami a otáčania, a 11,1 % pripadlo na jazdu v zlom pruhu a nesprávne predbiehanie.“ Autor článku ďalej uviedol, že jednou z možností ako znížiť počet nehôd a úmrtí na cestách je nehodám predchádzať (Karpat, 2007).

Ďalšou možnosťou by bolo zníženie počtu vozidiel. Keď sa však zamyslíme nad dnešnou dobou, tak klesanie, dokonca ani stagnácia tohto počtu sa nejaví ako reálna. V dnešnej dobe nie je nič nezvyčajné, ak napríklad 4 členná rodina vlastní aj 5 áut. Z toho nám vyplýva, že počet vozidiel na cestných komunikáciach bude stále len narastať. S tým však súvisia aj ďalšie problémy ako napríklad nadmerná produkcia CO2 alebo kolaps dopravy v každom väčšom meste. To je však nad rámec tejto témy.

V roku 2007 predstavil koncern General Motors telematický systém prenosu informácii medzi vozidlami. V tých časoch bolo cieľom systému V2V včas upozorniť posádku vozidla na nepredvídané nebezpečenstvo a v predstihu sa naň správne pripraviť a zareagovať. Medzi takéto nebezpečenstvá sa zaraďovalo riskantné konanie vodičov (odstavenie vozidla v zákrute, otáčanie sa za zákrutou), autá zoraďujúce sa za sebou pri spájaní viacerých pruhov do jedného, práce na ceste alebo kvôli zlým poveternostným podmienkam (námraza, sneh, hmla). Systém sústavy na komunikáciu V2V pozostávala z mikroprocesora, GPS prijímača a modulov realizujúcich bezdrôtovú komunikáciu. Táto sústava dokázala sledovať polohu vozidla, jeho rýchlosť a smer. Tieto údaje si potom vymieňala s okoloidúcimi vozidlami do vzdialenosti 2 až 5 kilometrov. Ak systém zistil neštandardnú dopravnú situáciu, odoslal informácie cez bezdrôtovoú sieť WLAN. Ostatné vozidlá v dosahu tieto informácie prijali a akusticky (cez reproduktory v aute) aj opticky (cez displej v aute) na ne posádku upozornili. Nie všetky informácie, ktoré vozidlo zachytilo, boli interpretované posádke. Dôvodom je, že nie všetky informácie sú v danom čase dôležité pre posádku. Systém mal upozorniť len na nebezpečné situácie na trase vozidla. Výhodou použitia bezdrôtovej siete WLAN bolo to, že táto technológia už existovala a nevyžadovala si žiadne náklady na prevádzku, čo platí dodnes. V roku 2009 mala byť Európskou úniou na účel komunikácie medzi vozidlami schválená jedna spoločná frekvencia (Karpat, 2007).

### 1.1.1 Skúšanie v praxi

Autor článku o V2V komunikácii medzi vozidlami (Karpat, 2007) v ňom vysvetľuje aj to, ako v tom čase to malo fungovať v praxi. Vysvetlil to na príklade hmly: „Ak vodič vojde do hmly, prvé, čo urobí zníži rýchlosť a zapne hmlové svetlá. Keď to zopakuje desať áut po sebe, dá sa táto informácia považovať za dôveryhodnú v porovnaní s tým, keď mladí zapnú hmlovky bezdôvodne len tak pre frajerinu.“ Inžinieri koncernu General Motors im umožnili si celý systém názorne vyskúšať. K dispozícii mali 4 vozidlá.

Prvá ukážka spočívala v tom, ako vie systém V2V upozorniť na blížiace sa policajné alebo záchranné vozidlo. Problém, ktorý nastáva a platí dodnes, je že sirénu síce počujeme, ale nemusíme vždy tušiť odkiaľ dané vozidlo prichádza, čo môže spôsobiť stres. Systém mal ukázať z ktorej strany sa dané vozidlo blíži a ako ďaleko je od Vás. Názorná ukážka je na nasledujúcom obrázku.



Obrázok Blížiace sa záchranné vozidlo

Druhá ukážka bolo stojace vozidlo v neprehľadnej zákrute alebo dopravné obmedzenie na neprehľadnom mieste. Aj keď prekážka nebola v našom jazdnom pruhu a nehrozila kolízia, tak systém V2V aj tak včas upozornil, akusticky aj vizuálne, že na ceste niečo stojí.

Ďalšia ukážka spočívala v tom, ako systém funguje pri vznikajúcej zápche na diaľnici. Systém na pomaly idúce alebo stojace vozidlo pred nami upozrnil najprv akusticky a vizuálne na displeji. Ak vodič nezareagoval na žiadne upozornenie, napríklad kvôli telefonovaniu, tak naše vozidlo poslalo informáciu vozidlu tomu pred ním, že sa blíži, a to zaplo brzdové svetlá.

Autor článku o V2V komunikácii medzi vozidlami (Karpat, 2007) ďalej vysvetľuje, že okrem upozorňovania vie V2V systém aj zasiahnuť v reálnom čase. Popisuje to na príklade, ak blížiace auto prichádzajúce do križovatky z bočnej cesty si Vás nevšíma a nespomaľuje, tak naše auto to zaregistruje a ešte pred vjazdom do križovatky prudko zabrzdí. Vďaka zájomnej komunikácii medzi sebou však môžu zastať obe autá súčasne.

### 1.1.2 Uvedenie do praxe

Dopísať...

## 1.2 V2V komunikácia v súčasnosti

Výhody/nevýhody/problémy a ich riešenia V2V

OBD

Raspberry Pi

## 1.3 Minipočítač Raspberry Pi

Bla bla bla



Obrázok Minipočítač Raspberry Pi 4

Na obrázku vyššie si môžete prezrieť ako daný minipočítač vyzerá. Na obrázku sa konkrétne nachádza 4. generácia tohto minipočítača.

### 1.2.1 Raspberry Pi 3

Stránka (thepihut.com, 2016) prezentuje Raspberry Pi 3 (model B) v poradí 3. generáciu tohto kultového minipočítača. Tak ako jeho predchodcovia, aj tento model má veľkosť kreditnej karty a môže byť použitý pre množstvo aplikácií.

Oproti jeho predchodcom ponúka väčší výkon a síce je 10-krát rýchlejší ako jeho 1. generácia. Taktiež má v sebe zabudovaný moduly pre bezdrôtovú sieť WLAN a Bluetooth.

Technická špecifikácia tohto minipočítača je nasledovná:

* čipset *Broadcom BCM2387*,
* 4-jadrový procesor s taktom 1,2 GHz a 64 bitovou architektúrou *ARM Cortex-A53*,
* zabudovaný modul pre bezdrôtovú sieť WLAN štandardu IEEE802.11/b/g/n,
* zabudovaný modul pre bezdrôtovú sieť Bluetooth 4.1,
* pamäť RAM veľkosti 1 GB,
* 4-krát USB 2.0 port,
* 4 pólový stereo výstup a kompozitný video port,
* HDMI výstup v HD kvalite,
* 10/100 Mbit Ethernet port,
* CSI port pre kameru,
* DSI port pre dotykový displej,
* port pre Micro SD kartu, ktorá slúži ako disk,
* napájací zdroj Micro USB,
* 40-pinový GPIO (General-Purpose Input/Output),

V rámci našej diplomovej práce používame práve túto generáciu minipočítača.

### 1.2.2 Raspberry Pi 4

V roku 2019 vyšla 4. generácia tohto minipočítača. Stránka (thepihut.com, 2019) uvádza viacero vylepšení oproti 3. generácii. A dosť výrazných. Medzi najhlavnejšie patrí zvýšenie taktu procesora na 1,5 GHz, pri pamäti RAM máme na výber 1 GB, 2 GB alebo 4 GB verziu, WLAN štandardu IEEE 802.11/a/c a Bluetooth 5.0, 10/100/1000 Mbit Ethernet port, 2 USB porty sú verzie 3.0 a mnoho ďalších.

## 1.4 OBD-II

Táto podkapitola poskytuje stručný prehľad:

* aplikácií, ktoré aktívne využívajú programovací jazyk Lua,
* modulov, ktoré môžu pomôcť pri vývoji aplikácie,
* online zdrojov a odbornej literatúry, odkiaľ možno čerpať ďalšie informácie,
* vývojových prostredí, pomocou ktorých možno urýchliť a zlepšiť samotný vývoj aplikácie.

### 1.4.1 Aplikácie

Programovací jazyk Lua využíva veľa komerčných aplikácií, ako aj aplikácií s otvoreným zdrojovým kódom. Na webovej stránke (lua-users.org, 2015) vieme nájsť prehľad tých najznámejších. Vyberieme aspoň niekoľko reprezentatívnych príkladov.

*SimCity 4* je hra s dlhoročnou tradíciou simulácie mesta. Hra využíva programovací jazyk Lua na konfiguráciu herných konštánt a na úpravu používateľského rozhrania.

*World of Warcraft* je hra, ktorá využíva programovací jazyk Lua v subsystéme používateľského rozhrania. Vďaka tomu sú používatelia schopní prispôsobiť si používateľské rozhranie prostredníctvom rôznych módov.

*CryENGINE2* je herný *engine*, na ktorom je postavená hra *Far Cry* a *Crysis*. Pomocou programovacieho jazyka Lua môžeme vylepšovať zbrane, prehrávať zvuky a pracovať s grafikou bez nutnosti použitia programovacieho jazyka C++.

Hra *Far Cry* používa programovací jazyk Lua na skriptovanie všetkých herných udalostí, umelej inteligencie a na editáciu hry v reálnom čase.

*Wireshark* je najpopulárnejší sieťový analyzátor. Pomocou programovacieho jazyka Lua môžeme napísať vlastný analyzátor paketových dát (*dissector* a *post-dissector*), ako aj mechanizmus na získanie dát z každého rámca (*tap*).

*FXCM Trading Station* je obchodná platforma. Programovací jazyk Lua sa tu využíva na vytváranie indikátorov a obchodných stratégií.

### 1.3.2 Moduly

Keďže programovací jazyk Lua poskytuje len základnú funkcionalitu, tak musí existovať spôsob, ako ho o ďalšiu funkcionalitu rozšíriť. Našli sme aplikáciu *LuaRocks* (*www.luarocks.org*), ktorá toto rieši. Aplikácia *LuaRocks* je balíčkový manažér, pomocou ktorého môžeme funkcionalitu programovacieho jazyka Lua rozšíriť o ďalšie moduly. Poskytuje nám obrovské množstvo modulov. My však predstavíme len zopár tých najužitočnejších.

Modul middleclass pridáva funkcionalitu pre objektovo orientované programovanie. Umožňuje nám jednoducho vytvárať triedy. Poskytuje nám aj koncept dedičnosti. (Rogalsky, 2012).

Modul busted je jednotkový testovací *framework*. Je veľmi dobre zdokumentovaný, flexibilný a plne vybavený. Podporuje aj alternatívne jazykové sady. Podporuje niekoľko štýlov tvrdení (*assertions*),objektovú introspekciu, *mocking* a *spying* objektov, odchyt chýb (Rogalsky, 2012).

Modul inspect je jednoduchý nástroj na vizualizáciu obsahu *tabuliek* ako aj ladenie aplikácie. Každý väčší projekt by mal takýto modul využívať (Rogalsky, 2012).

LuaSocket je modul, ktorý poskytuje podporu pre TCP a UDP transportné vrstvy. Podporuje protokoly ako SMTP, HTTP, FTP a mnoho ďalších (w3.impa.br, 2015).

Modul LuaFileSystem poskytuje prenositeľný spôsob prístupu k adresárovej štruktúre a súborovým atribútom (luarocks.org, 2016).

### Online zdroje

Na internete môžeme nájsť niekoľko užitočných zdrojov, z ktorých sa dajú efektívne čerpať informácie o programovacom jazyku Lua. Webová stránka *www.lua.org* je oficiálnou webovou stránkou o programovacom jazyku Lua. Môžeme na nej nájsť detailnejšie informácie o tomto jazyku, kompletnú dokumentáciu, novinky, zdrojové súbory a podobne. Na komunitnej webovej stránke *www.lua-users.org* môžeme nájsť wiki a archív mailovej komunikácie. Často kladené otázky môžeme nájsť na neoficiálnej webovej stránke *www.luafaq.org*. Veľmi dobre spracovaný tutoriál môžeme nájsť na webovej stránke *www.tutorialspoint.com/lua*.

### Odborná literatúra

Kníh k tejto téme môžeme nájsť viacero. My sme však vybrali iba dve, nakoľko ostatné nie sú najaktuálnejšie.

Kniha *Programming in Lua* (Ierusalimschy, 2013) poskytuje solídny základ pre tých, ktorí chcú začať aktívne používať programovací jazyk Lua. Táto kniha pokrýva všetky aspekty programovacieho jazyka Lua verzie 5 od základov až po API v programovacom jazyku C. Vysvetľuje, ako najlepšie využiť jeho vlastnosti a poskytuje množstvo vzorových príkladov. Kniha *Programming in Lua* je zameraná na ľudí aspoň s čiastočnými základmi v oblasti programovania, ale zároveň nepredpokladá žiadne predchádzajúce znalosti v programovacom jazyku Lua alebo v iných programovacích jazykoch. Tretie vydanie tejto knihy je zamerané na verziu 5.2. Všetky kapitoly spolu poskytujú viac ako 100 príkladov na precvičenie (thepihut.com, 2019).

Kniha *Lua Game Development Cookbook* (Kašuba, 2015) sprevádza čitateľa celým vývojom herného *engine*-u, pričom pomáha porozumieť, ako sú počítačové hry postavené. Začína s jednoduchými hernými konceptami použitými hlavne v 2D hrách, a následne sa presúva k pokročilejším 3D hrám. Obsahuje vyše 70 návodov. Odkrýva spôsoby, ako vyriešiť známe problémy v hernej mechanike. Učí o modernej počítačovej grafike, zvukoch, fyzike, umelej inteligencii a ďalších veciach. Obsahuje množstvo praktických príkladov, inštrukcií a vysvetlení, ktoré pomáhajú vytvoriť moderný herný *engine*. Od čitateľa sa vyžaduje znalosť programovacieho jazyka Lua (Kašuba, 2015).

### Vývojové prostredia

Vývojových prostredí, ktoré podporujú programovací jazyk Lua, existuje viacero. My spomenieme len tie, ktoré sme si sami odskúšali.

Editor *LuaEdit* podporuje verzie 5.1 a 5.2 programovacieho jazyka Lua. Podporuje „návrh na dokončenie“ príkazu, lokálne a vzdialené ladenie, zvýrazňovanie syntaxe, pokročilý manažment bodov prerušenia (*breakpoints*), výpis funkcií, výpis globálnych aj lokálnych premenných, a podobne. Licencovaný je pod GPL licenciou (luaedit.sourceforge.net, 2015).

*Eclipse* je IDE primárne určené pre programovací jazyk Java, ale pomocou modulu LDT (*Lua Development Tools*) je možné rozšíriť jeho podporu aj pre programovací jazyk Lua. Podporuje verzie 5.1 a 5.2. Tento editor je licencovaný pod EPL licenciou.

# 2 Ciele bakalárskej práce

V bakalárskej práci chceme predstaviť programovací jazyk Lua vývojárom softvéru, ktorí sa s týmto skriptovacím jazykom ešte nestretli, ale majú skúsenosti s inými vyššími programovacími jazykmi, prípadne so skriptovacími jazykmi pri tvorbe webových stránok (napr. Java, C, C++, PHP, JavaScript).

Hlavným cieľom bakalárskej práce je:

* preskúmať a popísať vlastnosti, koncepty a postupy typické pre programovanie v programovacom jazyku Lua,
* na základe získaných vedomostí a vlastných praktických skúseností vytvoriť kolekciu vzorových aplikácií, ktoré budú názorne demonštrovať použitie programovacieho jazyka Lua v praxi,
* poukázať na význam programovacieho jazyka Lua pre profesionálnu prax a sformulovať odporúčania pre záujemcov o použitie programovacieho jazyka Lua vo vlastných aplikáciách.

# 3 Analýza programovacieho jazyka Lua

V rámci tejto kapitoly rozoberieme vlastnosti, koncepty a postupy zaužívané v programovacom jazyku Lua. K jednotlivým podkapitolám sme vytvorili vzorové príklady, aby sme programovací jazyk Lua demonštrovali aj v praxi. Výnimkou je iba podkapitola *3.1 Základy*, nakoľko sa s vecami prezentovanými v tejto podkapitole budeme stretávať naprieč všetkými vzorovými príkladmi. Vzorový príklad ku každej podkapitole možno nájsť na priloženom CD v priečinku pomenovanom pr3\_*X*, kde *X* predstavuje číslo podkapitoly. V každom priečinku (okrem pr3\_9) sa nachádza súbor main.lua, ktorý slúži na spustenie vzorového príkladu. Predstavu o tom, ako vyzerá zdrojový kód v jazyku Lua, možno získať aj preštudovaním vzorového zdrojového kódu v prílohe A.

## 3.1 Základy

Každý kus zdrojového kódu, ktorý interpreter programovacieho jazyka Lua vykoná, sa nazýva *chunk*. Jednotlivé príkazy, podľa konvencie, by mali byť umiestnené na samostatnom riadku. Avšak, v jednom riadku možno mať aj dva a viac príkazov, ktoré by od seba mali byť oddelené bodkočiarkou. Nie je to ale nevyhnutné. Identifikátory môžu byť zložené z písmen, číslic a podtržítok, pričom nemôžu začínať číslicou. Existujú tzv. kľúčové slová, ktoré nemôžu byť použité ako identifikátory. Sú to: and, break, do, else, elseif, end, false, goto, for, function, if, in, local, nil, not, or, repeat, return, then, true, until a while. Identifikátory sú citlivé na veľkosť písmen. **true** je kľúčové slovo zatiaľ čo True a TRUE sú dva odlišné identifikátory. Podľa konvencie by sme sa mali vyhýbať tvorbe identifikátorov, ktoré začínajú podtržítkom nasledovaným jedným alebo viacerými veľkými písmenami (napr. \_G). Jednoriadkové komentáre je možné písať za dvojicu znakov --. Viacriadkové komentáre zasa medzi znaky --[[ a --]]. V programovacom jazyku Lua rozlišujeme lokálne a globálne premenné. Lokálne sú tie, ktoré sú deklarované pomocou kľúčového slova local. Globálne sú všetky ostatné. Dostupnosť lokálnych premenných je iba v rámci bloku, v ktorom sú deklarované, ako aj vo vnorených blokoch. Hodnoty do premenných a polí tabuliek priraďujeme pomocou znaku =. Možné je aj viacnásobné priradenie, kde premenné na ľavej strane sú oddelené čiarkou, tak isto, ako prislúchajúce hodnoty na pravej strane. Hodnoty, ktoré sú navyše na pravej strane, sa zahodia. Premenným, ktoré sú navyše na ľavej strane, sa priradí hodnota nil. Pokiaľ chceme nejakú časť zdrojového kódu oddeliť od zvyšku, resp. ju zvýrazniť, môžeme na to použiť blokovú konštrukciu do ... end. Každá bloková konštrukcia je v programovacom jazyku Lua ukončená kľúčovým slovom end. Pomocou funkcie require vieme načítať ďalší, resp. súvisiaci modul. Počet medzier, ktoré oddeľujú jednotlivé syntaktické prvky, je irelevantný. V niektorých prípadoch však musí byť prítomná aspoň jedna medzera.

### 3.1.1 Typy hodnôt

V programovacom jazyku Lua rozlišujeme 8 typov hodnôt: nil, boolean, number, string, table, function, userdata a thread. Typ hodnoty nil má iba jednu hodnotu, a to nil. Táto hodnota reprezentuje absenciu užitočnej hodnoty. Ak premenná alebo pole tabuľky reálne neexistuje, tak sa tvári ako keby existuje, pričom vráti práve túto hodnotu. Pomocou tejto hodnoty môžeme danú premennú alebo pole tabuľky vymazať z pamäte. Typ hodnoty boolean má iba dve hodnoty, a to true a false. Majú taký istý význam a použitie ako v iných programovacích jazykoch. Hodnota nil sa vyhodnocuje ako hodnota false. Napríklad, z programovacieho jazyka PHP môžeme byť zvyknutí, že hodnoty 0 a ““ (prázdny reťazec) sa vyhodnocujú tiež ako false, avšak v programovacom jazyku Lua to neplatí. Všetky hodnoty, okrem false a nil, sa teda vyhodnocujú ako hodnota true. Typ hodnoty number reprezentuje aj celé, aj reálne čísla. Samotné číslo je reprezentované 64 bitmi. Má teda dvojitú presnosť a pohyblivú rádovú čiarku (*floating point*). Pohyblivá rádová čiarka nasleduje štandard IEEE 754. Medzi platné čísla patrí napr. 10, 15.8, 9.22e-4 a 2.55e6 (2.55e+6). Na veľkosti písmena e nezáleží. Desatinná čiarka je reprezentovaná bodkou. Podporované sú aj hexadecimálne čísla, ktoré prefixujeme 0x (napr. 0xA1). Na veľkosti písmen X, A-F tiež nezáleží. Typ hodnoty string reprezentuje sekvencie znakov, tzv. reťazce. V reťazcoch možno uložiť znaky štandardu Unicode (UTF-8, UTF-16, ...) ako aj binárne dáta. Jednoriadkový reťazec sa píše buď medzi dvojicu dvojitých úvodzoviek alebo dvojicu jednoduchých úvodzoviek. Viacriadkový reťazec sa píše medzi znaky [[ a ]]. Pokiaľ do reťazca nevieme alebo nemôžeme priamo zapísať nejaký znak, môžeme si pomôcť znakom spätného lomítka, napr. \n pre nový riadok, \t pre tabulátor, \“ pre dvojitú úvodzovku, \‘ pre jednoduchú úvodzovku, \ddd a \xhh pre číselnú hodnotu, kde ddd je sekvencia maximálne troch decimálnych čísel, a kde hh je sekvencia presne dvoch hexadecimálnych čísel. Týchto špeciálnych znakov existuje ešte zopár. Dva reťazce je možné spojiť do jedného pomocou operátora ... Na reťazcoch, ktoré v sebe nesú číselné hodnoty, možno vykonávať aritmetické operácie. Typ hodnoty table reprezentuje asociatívne polia, tzv. tabuľky. Typ hodnoty function predstavuje funkcie. Typ hodnoty thread predstavuje korutiny. Posledným trom typom hodnôt sa detailnejšie venujeme v samostatných podkapitolách. Typ hodnoty userdata predstavuje dáta programovacieho jazyka C uložené v premenných programovacieho jazyka Lua. Takéto dáta môžu predstavovať napr. informácie o otvorenom súbore a pod.

### 3.1.2 Operátory

V programovacom jazyku Lua poznáme niekoľko operátorov. Medzi aritmetické operátory patrí sčítanie (+), odčítanie (-), násobenie (\*), delenie (/), *floor* delenie (//), umocňovanie (^), modulo (%) a negácia (-). Výsledkom operácií, ktoré používajú tieto operátory, nemusí byť len číselná hodnota. Medzi bitové operátory patrí bitový AND (&), bitový OR (|), bitový XOR (~), bitový NOT (~), posunutie bitov vľavo (<<) a posunutie bitov vpravo (>>). Výsledkom operácií, ktoré používajú tieto operátory, nemusí byť len číselná hodnota. Medzi porovnávacie operátory patrí *je menej ako* (<), *je viac ako* (>), *je menej ako alebo sa rovná* (<=), *je viac ako alebo sa rovná* (>=), *sa rovná* (==) a *sa nerovná* (~=). Výsledkom operácií, ktoré používajú tieto operátory, je hodnota true alebo false. Medzi logické operátory patrí and, or a not. Výsledkom operácií, ktoré používajú operátory and a or ale nie je hodnota true alebo false ako by sme čakali. Pri not je. Operátor and vráti prvý operand, ak sa vyhodnotí ako nepravdivý. V opačnom prípade vráti druhý operand. Operátor or vráti prvý operand, ak sa vyhodnotí ako pravdivý. V opačnom prípade vráti druhý operand. V prípade operátora and je druhý operand vyhodnotený iba, ak prvý operand sa vyhodnotil ako pravdivý. V prípade operátora or je druhý operand vyhodnotený iba, ak prvý operand sa vyhodnotil ako nepravdivý. Ďalším operátorom je operátor spojenia (..). Slúži hlavne na spojenie dvoch reťazcov do jedného. Môže byť ale použitý aj s inými typmi hodnôt. Operátor dĺžky (#) slúži na zistenie dĺžky reťazcov (počet bajtov) alebo tabuliek (počet prvkov indexovaných od indexu 1 po posledný index neobsahujúci hodnotu nil).

Tabuľka Priorita operátorov od najvyššej po najnižšiu

|  |  |
| --- | --- |
| **Operátory** | **Komentár** |
| ^ |  |
| not - ~ # | unárne operátory |
| \* / // % |  |
| + - |  |
| .. |  |
| << >> |  |
| & |  |
| ~ |  |
| | |  |
| < > <= >= == ~= |  |
| and |  |
| or |  |

### 3.1.3 Riadiace konštrukcie

Pokiaľ chceme otestovať výraz, či je pravdivý alebo nie, a podľa toho následne učiniť patričné kroky, tak máme k dispozícii konštrukciu if *podmienka* then ... else ... end, pričom časť else nie je povinná. K dispozícii máme aj konštrukciu elseif *podmienka* then ..., ktorú vkladáme za *if-then*. Na vytvorenie cyklu s neznámym počtom opakovaní máme k dispozícii konštrukcie while *podmienka* do ... end a repeat ... until *podmienka*. Cyklus while sa opakuje dovtedy, pokiaľ sa *podmienka* vyhodnocuje ako pravdivá. Cyklus repeat sa opakuje dovtedy, pokiaľ sa *podmienka* vyhodnocuje ako nepravdivá. Pri cykle repeat sa *podmienka* vyhodnocuje až po vykonaní jeho tela. Na vytvorenie cyklu so známym počtom opakovaní máme k dispozícii konštrukcie numerického a generického for. Konštrukcia numerického for cyklu sa zapisuje vo forme for *i* = *min*, *max*, *krok* do ... end alebo for *i* = *max*, *min*, -*krok* do ... end. *min* určuje minimálnu číselnú hranicu, *max* určuje maximálnu číselnú hranicu a *krok* určuje číslo, o ktoré sa premenná *i* zvýši alebo zníži každým opakovaním. *krok* je nepovinný, predvolená hodnota je 1. Treba mať na pamäti, že *min*, *max* a *krok* sa vyhodnotia iba raz, a to pred spustením cyklu. Premenná *i* môže mať aj iný názov. Generický for cyklus slúži na prechádzanie hodnôt vrátených iteračnou funkciou. Jeho konštrukcia sa zapisuje vo forme for *vars* in *iter*, *args* do ... end. *iter* predstavuje iteračnú funkciu. args predstavuje voliteľné argumenty oddelené čiarkou, ktoré sa jej odovzdajú. Často sa na mieste *iter*, *args* volajú funkcie pairs a ipairs, ktoré túto iteračnú funkciu vracajú vrátane tých argumentov. *vars* predstavuje zoznam premenných oddelených čiarkou, ku ktorým sa priradia hodnoty vrátené iteračnou funkciou pri každom opakovaní. Premenným, ktoré sú navyše, sa priradí hodnota nil. Pokiaľ chceme cyklus prerušiť, môžeme tak učiniť pomocou príkazu break, prípadne return.

## 3.2 Dátové štruktúry

Pokiaľ porovnáme dostupné API programovacích jazykov Java a Lua, tak zistíme, že API programovacieho jazyka Java nám poskytuje niekoľko dátových štruktúr, ako napr. zoznam (*list*), mapu, set a ďalšie, zatiaľ čo API programovacieho jazyka Lua ani jednu. Oba jazyky nám však natívne poskytujú tzv. polia (*arrays*). V programovacom jazyku Lua sa označujú skôr ako tabuľky (*tables*). Tabuľky sú však mocnejší nástroj ako polia v programovacom jazyku Java. Spomínané dátové štruktúry môžu byť jednoducho implementované práve pomocou tabuliek. V prvom rade si preto rozoberieme samotné tabuľky.

### 3.2.1 Tabuľky

Tabuľky možno definovať ako asociatívne polia. To znamená, že jednotlivé hodnoty uložené v tabuľke môžu byť indexované rôznymi hodnotami (okrem hodnoty nil), pričom najčastejšie sú to celočíselné a reťazcové hodnoty. Tabuľky predstavujú objekty, pretože každá má jedinečnú identitu. Platí to aj v prípade, ak dve a viac tabuliek sú hodnotovo identické. Novú prázdnu tabuľku možno vytvoriť konštrukciou {}. Medzi tieto zátvorky možno zapísať aj počiatočné hodnoty oddelené čiarkou. Pokiaľ indexy nie sú uvedené explicitne, tak sú uvedené implicitne ako celočíselné hodnoty začínajúce od 1. Index[[1]](#footnote-1) a hodnota sa medzi sebou oddeľujú znakom =. Takýto pár sa označuje ako kľúč-hodnota (*key-value*), resp. pole. Pokiaľ ako index chceme použiť hodnotu premennej alebo nejaký iný výraz, tak ho musíme obaliť hranatými zátvorkami. Pokiaľ ako index chceme použiť reťazec, tak ho nemusíme obalovať ani do hranatých zátvoriek, ani do úvodzoviek, v prípade, že sa jedná o platný identifikátor. Ak chceme získať nejakú hodnotu z tabuľky, tak za premennú, držiacu danú tabuľku, doplníme hranaté zátvorky, a medzi ne vložíme hodnotu reprezentujúcu index. Ak chceme získať hodnotu, ktorej index je reťazec, a ktorý je zároveň aj platným identifikátorom, tak ho môžeme zapísať, bez úvodzoviek, za premennú držiacu danú tabuľku a bodku (napr. namiesto t[“name“] napíšeme t.name). Ak sa pokúsime získať hodnotu neexistujúceho indexu, tak dostaneme hodnotu nil. Tabuľky sú dynamické, vždy rastú podľa našich potrieb. Pokiaľ chceme pridať novú hodnotu do tabuľky alebo prepísať tú aktuálnu, tak použijeme takú istú notáciu ako pri získaní hodnoty z tabuľky, pričom samotnú hodnotu priradíme pomocou znaku =. Naopak, ak chceme nejakú hodnotu z tabuľky vymazať, tak danému indexu priradíme hodnotu nil. Dĺžku tabuľky vieme zistiť cez operátor dĺžky (#), napr. #t. Pri tomto operátore je dôležité si uvedomiť, že vrátená dĺžka reprezentuje iba celočíselné indexy od indexu 1 po posledný index, ktorého hodnota nie je nil. Z toho vyplýva, že ak máme v tabuľke hodnotu *a* na indexe 1 a hodnotu *b* na indexe 3, pričom hodnota na indexe 2 je implicitne alebo explicitne nil, tak tento operátor nám vráti dĺžku *1*.

### 3.2.2 Polia

Pole možno charakterizovať ako kolekciu hodnôt, ktorá je indexovaná celočíselnými hodnotami. Pole vytvoríme tak, že medzi zložité zátvorky umiestnime hodnoty oddelené čiarkou (napr. {“a“, “b“}). Ako sme spomínali, indexovanie začína od 1. Hodnota *a* sa nachádza na indexe 1, zatiaľ čo hodnota *b* na indexe 2. Kľudne by sme tieto indexy mohli v danej tabuľke uviesť, avšak bola by to práca navyše. Indexovanie môže začínať aj od iného čísla ako 1, napr. od -1. Takéto pole potom vytvoríme nasledovne: {[-1] = “x“, [0] = “y“, “a“, “b“}. Problém ale nastáva s operátorom dĺžky, ktorý namiesto *4* vráti *2*. Bez problémov je možné vytvoriť aj viac-dimenzionálne pole. Riešením je tabuľka v tabuľke.

### 3.2.3 Mapy

Mapa ako taká je založená na kľúčoch, ku ktorým je priradená nejaká hodnota. Každý kľúč sa v mape môže nachádzať iba raz. V programovacom jazyku Lua môže mapa vyzerať nasledovne: {name = “Dávid“, [“last name“] = “Bojnanský“}. Operátor dĺžky v tomto prípade vracia *0*.

### 3.2.4 Sety

Set možno charakterizovať ako kolekciu jedinečných hodnôt. To znamená, že každá hodnota sa v danej kolekcii môže nachádzať iba raz. V programovacom jazyku Lua to dosiahneme tak, že ako kľúč použijeme danú hodnotu, ku ktorému priradíme hodnotu true (napr. {[{}] = true, [101] = true, [“Dávid“] = true}). Operátor dĺžky v tomto prípade vracia *0*. Namiesto hodnoty true možno použiť aj inú hodnotu (okrem hodnoty nil), ako napríklad číslo, ktoré by virtuálne indikovalo počet hodnôt v danej kolekcii.

### 3.2.5 Dvojkoncové fronty

Dvojkoncový front možno charakterizovať ako kolekciu hodnôt, do ktorej pridávame hodnotu na jeden z koncov, získavame hodnotu z jedného z koncov, odstraňujeme hodnotu z jedného z koncov. S hodnotami medzi oboma koncami nemanipulujeme. Tá istá hodnota sa v dvojkoncovom fronte môže nachádzať aj viackrát. Dvojkoncový front indexujeme celočíselnými hodnotami od 1. Hodnotu pridávame na jeden z koncov pomocou funkcie table.insert, ktorá v prípade, že pridáme hodnotu na začiatok (index 1), zvýši všetky ostatné indexy o jedna. Hodnotu odstraňujeme z jedného z koncov pomocou funkcie table.remove, ktorá v prípade, že odstránime hodnotu zo začiatku (index 1), zníži všetky ostatné indexy o jedna. Hodnoty z dvojkoncového frontu získavame pomocou indexov 1 a #deque, kde deque predstavuje tabuľku reprezentujúcu daný dvojkoncový front.

### 3.2.6 Spájané zoznamy

Spájaný zoznam (*linked list*) sa podobá polu. Rozdiel je v tom, že hodnota sa neukladá do tabuľky pod index, ale do uzla. Každá hodnota je uložená v jednom uzli. Tento uzol sa potom odkazuje na predchádzajúci a/alebo nasledujúci uzol, v prípade, že sú dostupné. Každý uzol je reprezentovaný tabuľkou, ktorá obsahuje polia ako value, next a prev.

### Vzorový príklad pr3\_2

Aby sme jednotlivé dátové štruktúry demonštrovali v praxi, naprogramovali sme jednoduchý program, ktorý je založený na výpise stavov. Okrem neho sme však museli naprogramovať aj niektoré zo spomínaných dátových štruktúr. Pri nich stojí za povšimnutie spôsob, akým sú naprogramované. Za povšimnutie stojí aj spôsob, akým sa pracuje so samotnými tabuľkami, ktorý môžeme vidieť hneď na začiatku súboru main.lua.

Tabuľka Zoznam súborov vzorového príkladu pr3\_2

|  |  |
| --- | --- |
| **Súbor** | **Popis** |
| main.lua | spúšťací súbor programu |
| set.lua | obsahuje definíciu triedy setu |
| deque.lua | obsahuje definíciu triedy dvojkoncového frontu |
| linked\_list.lua | obsahuje definíciu triedy spájaného zoznamu |

## 3.3 Funkcie

Funkcie sú v programovacom jazyku Lua prvotriedne (*first-class*). To znamená, že funkcia je vlastne hodnota typu function, ktorá môže byť uložená v premennej alebo v poli tabuľky, môžeme ju odovzdať ako argument inej funkcii, môžeme ju vrátiť pomocou príkazu return vo funkcii alebo v súbore. K premenným, deklarovaným pomocou kľúčového slova local v rámci funkcie, je možné pristupovať v rámci danej funkcie, ako aj jej vnorených (anonymných) funkcií. Z vonku sú tieto premenné nedostupné. Premenné, ktoré nie sú v rámci funkcie deklarované pomocou kľúčového slova local, sú globálne. Výnimkou sú iba parametre funkcie. Tie sú implicitne označené kľúčovým slovom local. V prípade, ak v rámci funkcie vytvoríme novú globálnu premennú, tak k nej budeme môcť pristupovať aj zvonku, avšak až po zavolaní danej funkcie.

### 3.3.1 Definovanie funkcie

Keďže funkcia je hodnota typu function uložená v premennej, tak funkciu vieme definovať nasledovne:

local f -- ak sa jedná o lokálnu funkciu

f = function (a, b, c) ... end

Pokiaľ sa jedná o lokálnu funkciu, tak deklarácia lokálnej premennej f musí byť na samostatnom riadku pred definíciou funkcie v prípade, ak vo funkcii f môže dôjsť k rekurzívnemu volaniu. V opačnom prípade môžeme kľúčové slovo local presunúť pred f = ... Medzi okrúhle zátvorky uvádzame parametre funkcie oddelené čiarkou. Či existuje nejaký limit na ich počet, sme nezistili. Z niektorých programovacích jazykov môžeme byť zvyknutí na parametre, ktoré majú priradené predvolené hodnoty. V programovacom jazyku Lua túto možnosť nemáme. Tento problém sa ale rieši v rámci tela funkcie. Vyššie uvedený spôsob definície funkcie sa ale v praxi príliš často nepoužíva. Namiesto neho sa používa častejšie tento spôsob:

local function f(a, b, c) ... end

Táto definícia je totožná s tou vyššie. Kľúčové slovo local nám vypadáva, pokiaľ sa nejedná o lokálnu funkciu. Výsledok funkcie vraciame pomocou príkazu return. Tento príkaz ale nie je povinný. V prípade, že ho vynecháme, tak daná funkcia sa potom správa tak, ako keby vracala hodnotu nil.

### 3.3.2 Volanie funkcie

Ak máme definovanú funkciu f, ktorá prijíma tri celočíselné argumenty, tak ju vieme zavolať nasledovne: f(1, 2, 3). Pokiaľ nejaká funkcia prijíma iba jeden argument, ktorý je typu string alebo table, tak okrúhle zátvorky môžeme kľudne vynechať (napr. print “hello“ alebo table.sort {“b“, “a“}). Funkcii, pri jej volaní, nemusíme nutne odovzdať presne taký istý počet argumentov, aký má vo svojej deklarácii. Môžeme jej kľudne odovzdať aj menej alebo aj viac. V prípade, že jej odovzdáme viac ako očakáva, tak argumenty, ktoré sú navyše, sa jednoducho zahodia. V prípade, že jej predáme menej ako očakáva, tak argumenty, ktoré chýbajú, sa implicitne nastavia na hodnotu nil. Argumenty s hodnotou nil je nutné v rámci tela funkcie nejako ošetriť. Napríklad, ak sa jedná o povinný argument, a tento argument chýba, resp. má priradenú hodnotu nil, tak vyvoláme chybu. Naopak, ak sa jedná o nepovinný argument, tak hodnotu nil nahradíme za nejakú predvolenú hodnotu.

### 3.3.3 Pomenované argumenty

Ak definujeme funkciu, ktorá prijíma veľa argumentov, tak problém môže nastať v tom, že si nebudeme pamätať poradie, v akom tieto argumenty máme danej funkcii odovzdať pri volaní. Programovací jazyk Lua nám natívne neumožňuje pomenovať jednotlivé argumenty pri volaní, ako nám to umožňuje napríklad programovací jazyk Ruby. Riešením tohto problému môže byť tabuľka. Funkciu jednoducho definujeme tak, že bude prijímať iba jeden argument, a to tabuľku. Pri volaní jej predáme ako argument tabuľku, ktorá bude obsahovať hodnoty priradené k jednotlivým kľúčom, reprezentujúce názvy parametrov funkcie. Názorné volanie funkcie f, ktorá prijíma tri argumenty, by vyzeralo nasledovne: f {b = 2, c = 3, a = 1}.

### 3.3.4 Variabilný počet argumentov

Programovací jazyk Lua nám ponúka jeden špeciálny parameter pomenovaný .... Tento parameter, ak ho použijeme, musí byť vždy uvedený ako posledný parameter v zozname parametrov. Názorná definícia funkcie f by s týmto parametrom vyzerala nasledovne:

local function f(a, ...) ... end

Ako sme spomínali, funkcii pri volaní môžeme odovzdať viac argumentov, ako očakáva. Problém ale nastavá vtedy, ak chceme nejakým spôsobom pracovať aj s týmito argumentami. Rovno si to vysvetlime na príklade. Predstavme si, že funkcia očakáva od 1 000 do 2 000 argumentov. Definovať funkciu s 2 000 parametrami je neefektívne. A tento problém rieši práve parameter .... Tento parameter reprezentuje všetky odovzdané argumenty od miesta, kde je deklarovaný. Hodnota argumentu ... ale nie je tabuľka. Ak funkcii f predáme argumenty 1, 2 a 3, tak hodnota argumentu ... bude 2 a 3. Pokiaľ cheme z argumentu ... spraviť tabuľku, tak ho jednoducho ohraničíme zloženými zátvorkami: {...}. Ak teda hodnota argumentu ... je 2 a 3, tak jeho tabuľka bude vyzerať nasledovne: {2, 3}. Argument ... môžeme predať opäť ako argument ďalšej funkcii. Ak máme napríklad funkciu add, ktorá prijíma dva argumenty a a b, a ktorej úloha je oba argumenty spočítať, tak by sme túto funkciu mohli zavolať vo forme add(...), kde ... reprezentuje stále hodnoty 2 a 3. Výsledok tohto volania by bol 5. Ak by sme ale funkciu add zavolali napríklad takto:

add(..., 10) -- to isté ako: add(2, 10); hodnota 3 je zahodená

alebo

add(10, ...) -- to isté ako: add(10, 2, 3); hodnota 3 je zahodená

tak výsledok týchto volaní by bol v oboch prípadoch 12. Toto isté pravidlo platí aj v ďalších aspektoch programovacieho jazyka Lua, ako napríklad pri priraďovaní hodnoty do premennej a pod.

### 3.3.5 Viacnásobný výsledok

Funkcie môžu pomocou príkazu return vrátiť viac ako len jednu hodnotu. Jednotlivé hodnoty oddeľujeme čiarkou. Počet vrátených hodnôt môže byť aj variabilný. Ak máme definovanú funkciu f, ktorá vracia hodnoty 1, 2 a 3, tak tieto hodnoty získame nasledovným spôsobom: local a, b, c = f(). Ak by sme mali ale takýto príkaz: local a, b, c = f(), 10, tak premenná a by obsahovala hodnotu 1, premenná b by obsahovala hodnotu 10 a premenná c by obsahovala hodnotu nil. Iná situácia by opäť nastala pri takomto príkaze: local a, b, c = 10, f(). Premenná a by obsahovala hodnotu 10, premenná b by obsahovala hodnotu 1 a premenná c by obsahovala hodnotu 2. Hodnota 3 by bola zahodená. Táto vlastnosť je totožná s tou pri parametri ....

### 3.3.6 Anonymné funkcie

Jedná sa o funkcie deklarované bez mena. Ak sa nad tým zamyslíme, tak v podstate všetky funkcie v programovacom jazyku Lua sú anonymné. Meno získavajú vtedy, keď sú priradené do premennej alebo do poľa tabuľky.

### Vzorový príklad pr3\_3

Vzorová aplikácia, ktorú sme naprogramovali, demonštruje spôsoby ako definovať a volať funkcie všeobecne, ako definovať a volať funkcie s nepovinnými parametrami, s pomenovanými parametrami, s variabilným počtom argumentov. Ďalej nám ukazuje spôsoby ako vracať jeden výsledok alebo viacnásobný výsledok, ako tieto výsledky získať a pod.

## 3.4 Korutiny

Autor (Ierusalimschy, 2013) popisuje korutiny ako funkcionalitu podobnú vláknam, ktoré môžeme poznať z iných programovacích jazykov. Korutina predstavuje postupnú exekúciu zdrojového kódu, s vlastným zásobníkom, s vlastnými lokálnymi premennými, a s vlastným inštrukčným ukazovateľom. Hoci korutina má vlastné lokálne premenné, tak má prístup aj k tým globálnym. Hlavným rozdielom medzi korutinami a vláknami je, že vlákna bežiaceho programu sa v danom čase vykonávajú paralelne, zatiaľ čo pri korutinách sa vykonáva iba jedna korutina v danom čase. Korutiny sa preto označujú ako kolaboratívne. Vykonávanie korutiny je prerušené len v prípade, ak o to explicitne požiadame.

To, že v danom čase sa vykonáva iba jedna korutina ale neznamená, že korutiny musia bežať iba sekvenčne. Korutiny môžu bežať aj paralelne, a to tak, že prerušíme aktuálne bežiacu korutinu a spustíme nejakú inú. Výhodou korutín oproti vláknam je, že sa nemusíme starať o synchronizáciu, pretože nedochádza k súbehu. Naopak, pri zlom riadení korutín, sa aplikácia môže javiť ako spomalená.

Programovací jazyk Lua poskytuje pre prácu s korutinami modul coroutine, ktorý obsahuje viacero funkcií. Nájsť tu môžeme funkcie ako create, wrap, resume, yield, status, running a isyieldable.

Funkcia create po zavolaní vytvorí a vráti novú korutinu. Vrátená korutina je typu thread. Funkcia create prijíma iba jeden argument, a to funkciu (vo väčšine prípadov sa jedná o anonymnú funkciu) nazývanú *body*. *Body* funkcia reprezentuje zdrojový kód, ktorý daná korutina má vykonať. Vrátená korutina nie je automaticky spustená, ale namiesto toho sa nachádza v stave *prerušená* (*suspended*).

Funkcia wrap po zavolaní vytvorí novú korutinu tak ako create. Rozdiel je ale v tom, že funkcia wrap *vracia* funkciu, ktorú vždy keď zavoláme, tak zakaždým spustí/obnoví danú korutinu pomocou funkcie resume, ktorej výsledok vráti bez prvej *boolean* hodnoty. *Vrátenej* funkcii pri volaní možno odovzdať aj extra argumenty, ktoré sa následne odovzdajú, ako extra argumenty, funkcii resume. Ak dôjde k chybe v rámci danej korutiny, tak sa táto chyba, namiesto jej zachytenia, len spropaguje ďalej. Keďže funkcia wrap nevracia korutinu, tak nemáme ako zistiť jej stav pomocou funkcie status.

Funkcia resume pri zavolaní spustí/obnoví danú korutinu. Daná korutina prechádza zo stavu *prerušená* do stavu *bežiacia* (*running*). Ako prvý argument prijíma práve tú korutinu, ktorá má byť spustená/obnovená. Funkcii resume možno odovzdať aj extra argumenty, ktoré sa odovzdajú *body* funkcii, ak korutina ešte nebola spustená. Naopak, ak korutina už spustená bola, ale došlo k jej prerušeniu pomocou funkcie yield, tak sa tieto odovzdané extra argumenty vrátia ako výsledok volania funkcie yield. Funkcia resume vracia hodnotu true a extra argumenty, ktoré boli odovzdané cez funkciu yield, alebo, ktoré boli vrátené príkazom return v *body* funkcii. Všetko za predpokladu, že nedošlo k chybe. Ak k chybe došlo, tak funkcia resume vracia hodnotu false a chybovú správu. Ak korutina spustí/obnoví inú korutinu, tak prechádza zo stavu *bežiaca* do stavu *normálna* (*normal*). To znamená, že korutina je síce aktívna, ibaže sa nevykonáva. Korutina, ktorej *body* funkcia bola úspešne vykonaná sa už nedá obnoviť. Ak sa o to pokúsime, tak obdržíme ako výsledok hodnotu false a chybovú hlášku.

Funkcia yield po zavolaní preruší aktuálne sa vykonávajúcu korutinu. Tejto funkcii preto neodovzdávame ako argument korutinu, ktorú chceme prerušiť. Ako sme už pri funkcii resume spomínali, funkcii yield možno odovzdať extra argumenty, ktoré budú vrátené funkciou resume.

Funkcia status po zavolaní vráti aktuálny stav tej korutiny, ktorú sme jej odovzdali ako argument. Korutina môže byť v jednom z týchto stavov: *prerušená* (*suspended*), *bežiaca* (*running*), *normálna* (*normal*) a *mŕtva* (*dead*). Stavy *prerušená*, *bežiaca* a *normálna* sme priebežne vysvetlili. Stav *mŕtva* reprezentuje taký stav, kedy korutina úspešne vykonala celú svoju *body* funkciu, alebo pokiaľ korutina bola zastavená v dôsledku chyby.

Funkcia running po zavolaní vráti aktuálne sa vykonávajúcu korutinu a hodnotu true, pokiaľ je daná korutina označená ako hlavná. Táto funkcia neprijíma žiadne argumenty.

Funkcia isyieldable po zavolaní vráti *boolean* hodnotu. Hodnotu true, pokiaľ aktuálne sa vykonávajúca korutina môže byť prerušená pomocou funkcie yield. Hodnotu false v opačnom prípade. Táto funkcia neprijíma žiadne argumenty.

Uplatnenie korutín môžeme nájsť napríklad vo vzore *producer-consumer*, môžeme pomocou nich implementovať iterátory pre generický for cyklus, alebo ich môžeme využiť na paralelnú prácu so súbormi, ako aj paralelné sťahovanie súborov z internetu.

### Vzorový príklad pr3\_4

Aby sme demonštrovali prácu s korutinami, vytvorili sme aplikáciu, ktorej cieľom je poskladať básničku z dvoch textových súborov, pričom v jednom súbore sú riadky nepárne a v druhom tie párne. Prvý výstup aplikácie nám ukazuje, aký výsledok obdržíme, ak korutiny nepoužijeme, a druhý ak áno. Za povšimnutie stoja prezentované funkcie, ktoré sme v danej aplikácií použili.

Tabuľka Zoznam súborov vzorového príkladu pr3\_4

|  |  |
| --- | --- |
| **Súbor** | **Popis** |
| main.lua | spúšťací súbor programu |
| with.lua | obsahuje verziu aplikácie s použitím korutín |
| without.lua | obsahuje verziu aplikácie bez použitia korutín |
| poem-even-rows.txt | obsahuje párne riadky básničky |
| poem-odd-rows.txt | obsahuje nepárne riadky básničky |

## 3.5 Chyby

Programovací jazyk Lua býva najčastejšie vložený do hostiteľskej aplikácie, a preto nemôže jednoducho ukončiť alebo *zhodiť* danú aplikáciu, pokiaľ dôjde k chybe. Namiesto toho, vždy keď dôjde k chybe, tak sa iba jednoducho ukončí aktuálny *chunk* a riadenie sa prenechá hostiteľskej aplikácii (Ierusalimschy, 2013).

V programovacom jazyku Lua rozlišujeme dva typy chýb:

* syntaktické, ku ktorým dochádza pri nesprávnom použití jazykových prvkov, ako napríklad operátorov, výrazov, jazykových konštrukcií a pod. Tieto chyby sa ošetrujú oveľa jednoduchšie ako tie *runtime-ové*. Tieto chyby nám môže odhaliť aj textový editor.
* *runtime-ové*, ku ktorým dochádza počas behu programu. V tomto prípade môže byť dôvodov nespočet, ako napríklad chybný vstup, volanie funkcie na hodnote, ktorá funkciu nepredstavuje, neexistujúci súbor, odovzdaný nevhodný argument volaniu funkcie a pod.

V programovacom jazyku Lua môžeme programátorovi o neočakávanej situácii dať vedieť dvoma spôsobmi. Prvým je, že vyvoláme chybu pomocou funkcie error. Toto správanie môžeme nájsť napríklad u funkcii require. Druhým, častejšie preferovaným spôsobom je, že ak vo funkcii dôjde k nejakej neočakávanej situácii, tak pomocou príkazu return vrátime kód chyby (najčastejšie hodnotu nil), nasledovaný chybovou správou. Toto správanie môžeme nájsť napríklad u funkcii io.open. Neexistujú presne definované pravidlá, ktorý spôsob kedy použiť. Je to skôr o preferencii, prípadne nejakých interných pravidlách. Avšak môžeme nájsť aj nepísané pravidlo, ktoré hovorí, že ak neočakávanú situáciu vieme ľahko vyriešiť, tak by sme mali použiť prvý spôsob. V opačnom prípade ten druhý. Na prácu s chybami máme k dispozícii dve funkcie: error a assert. Ako sme už spomínali, funkciou error vyvoláme chybu, pričom ukončíme aktuálny *chunk*. Funkcia error prijíma dva argumenty. Prvým je správa, ktorá má vo väčšine prípadov textovú podobu (reťazec). Môže mať aj podobu tabuľky (napr. {code=500, msg=“chyba servera“}). Druhým je pozícia chyby. Jedná sa o nepovinný argument s predvolenou hodnotou 1. Hodnota 1 nám určuje miesto v súbore, kde funkcia error bola zavolaná. Hodnota 2 nám určuje miesto v súbore, kde bola zavolaná funkcia, ktorá v sebe zavolala funkciu error. A takto to pokračuje ďalej. Funkcia assert zavolá funkciu error v prípade, že jej prvý argument je vyhodnotený ako nepravdivý (hodnota argumentu je teda nil alebo false). Ako druhý argument jej môžeme odovzdať správu, ktorá sa odovzdá funkcii error. Jedná sa o nepovinný argument, ktorý keď vynecháme, tak sa použije predvolená správa “assertion failed!“. V prípade, že jej prvý argument sa vyhodnotí ako pravdivý, tak ho funkcia assert vráti. V praxi vyzerá použitie funkcie assert napríklad aj takto: n = assert(io.read(“\*n“), “vstupná hodnota nie je číslo“).

### 3.5.1 Ošetrenie chýb

Chyby vyvolané funkciou error môžeme ošetriť pomocou funkcií pcall alebo xpcall, čím zamedzíme ukončeniu aktuálneho *chunk-u*. Funkcia pcall prijíma ako argument (anonymnú) funkciu *f*, ktorá môže vyvolať chybu pomocou funkcie error. Funkcii pcall možno odovzdať aj extra argumenty, za prvý argument, ktoré sa odovzdajú funkcii *f* pri volaní. Ak funkcia *f* nevyvolá žiadnu chybu, tak funkcia pcall vráti hodnotu true, nasledovanú vrátenými hodnotami funkcie *f*. V opačnom prípade vráti hodnotu false a chybovú správu. Ako sme spomínali, funkcii error možno odovzdať ako chybovú správu napríklad aj tabuľku. K tejto tabuľke potom pristupujeme klasicky pomocou kľúčov. Funkcia xpcall funguje na ten istý princíp. Rozdiel je ale v tom, že za funkciou *f*, teda ako druhý argument, prijíma chybového ošetrovateľa (*error handler).* Extra argumenty možno odovzdať, tým pádom, až za ním. Ako chybového ošetrovateľa možno použiť existujúce funkcie debug.debug alebo debug.traceback, prípadne si napísať vlastného.

### Vzorový príklad pr3\_5

Naša vzorová aplikácia demonštruje oba predstavené spôsoby ako chybu vyvolať a ako ju aj ošetriť. Za povšimnutie stojí aj spôsob, ako ošetriť nevhodné argumenty.

## 3.6 Metatabuľky a metametódy

Vo väčšine prípadov má každá hodnota v programovacom jazyku Lua predvídateľnú sadu operácií. Číselné hodnoty môžeme spočítať, reťazce môžeme spojiť, páry kľúč-hodnota môžeme vložiť do tabuľky a pod. Avšak, nemôžeme spočítať tabuľky, nemôžeme porovnať funkcie, nemôžeme vykonať zavolanie na reťazec a pod. Pokiaľ nepoužijeme metatabuľky. Metatabuľky nám umožňujú meniť správanie hodnoty, ak je konfrontovaná s nedefinovanou operáciou. Napríklad, ak použijeme metatabuľky, tak môžeme definovať správanie ako sa má vypočítať výraz a + b, kde a aj b sú tabuľky. Vždy, keď sa snažíme spočítať dve tabuľky, tak sa skontroluje, či aspoň jedna z nich má definovanú metatabuľku, a či táto metatabuľka obsahuje pole \_\_add. Ak sa toto pole nájde, tak sa zavolá jeho korešpondujúca hodnota, označovaná ako metametóda, ktorá vykoná operáciu spočítania. Každá hodnota v programovacom jazyku Lua môže mať pridruženú metatabuľku (Ierusalimschy, 2013).

Metatabuľka nie je nič viac než klasická tabuľka, ktorá obsahuje kľúče reprezentujúce názvy metametód začínajúce \_\_. Avšak, metatabuľka môže byť aj prázdna. Metametóda býva vo väčšine prípadov (anonymná) funkcia, ale nemusí to tak byť vždy. Hodnoty typu table a userdata používajú individuálne metatabuľky, zatiaľ čo hodnoty ostatných typov používajú zdieľanú metatabuľku naprieč všetkými hodnotami daného typu. Z toho vyplýva, že reťazce “hello“ a “world“ majú rovnakú metatabuľku, číselné hodnoty 10 a 20 majú tiež rovnakú metatabuľku, avšak odlišnú od tej, ktorú majú reťazce, zatiaľ čo tabuľky {5}, {10} a {10} môžu mať metatabuľky rovnaké (teda zdieľané) alebo odlišné medzi sebou. Ako sa z nášho demonštratívneho programu môžete dozvedieť, tak štandardne majú nastavenú metatabuľku iba reťazce a hodnoty typu userdata. Dokonca aj novovytvorená tabuľka nemá nastavenú žiadnu metatabuľku. Avšak, prostredníctvom programovacieho jazyku Lua môžeme manipulovať s metatabuľkami iba v tabuľkách. V opačnom prípade musíme použiť programovací jazyk C. Metatabuľku môžeme tabuľke nastaviť pomocou funkcie setmetatable, kde ako prvý argument jej odovzdávame tabuľku, ktorej chceme nastaviť metatabuľku uvedenú v druhom argumente. Metatabuľku určitej tabuľky môžeme získať pomocou funkcie getmetatable, kde ako argument odovzdávame tabuľku, ktorej metatabuľku chceme získať. Programovací jazyk Lua poskytuje aj funkcie rawset, rawget, rawequal a rawlen. Funkcia rawset slúži na obídenie metametódy \_\_newindex a prijíma tri argumenty. Ako prvý argument jej odovzdávame tabuľku, ktorej chceme nastaviť reálny kľúč (druhý argument) na určitú hodnotu (tretí argument). Funkcia rawget slúži na obídenie metametódy \_\_index a prijíma dva argumenty. Ako prvý argument jej odovzdávame tabuľku, z ktorej chceme získať skutočnú hodnotu priradenú ku kľúču uvedenom ako druhý argument. Funkcia rawequal slúži na obídenie metametódy \_\_eq a prijíma dva argumenty, ktoré sa majú porovnať, či sú identické (podľa toho vracia hodnotu true alebo false). Funkcia rawlen slúži na obídenie metametódy \_\_len a prijíma jeden argument, a to buď tabuľku alebo reťazec. Vracia „skutočnú“ dĺžku odovzdaného objektu. Keď programovací jazyk Lua hľadá metametódu podľa typu výrazu, uskutoční nasledujúce kroky: ak prvá hodnota výrazu má metatabuľku s príslušnou metametódou, tak sa použije tá; ak ju prvá nemá a existuje druhá, ktorá ju má, tak sa použije tá z druhej hodnoty; ak ju nemá ani jedna, tak sa vyvolá chyba.

V programovacom jazyku Lua máme k dispozícii niekoľko metametód. Rozdeliť ich môžeme do viacerých kategórií: aritmetické, bitové, porovnávacie, tabuľkové a ostatné.

### 3.6.1 Aritmetické metametódy

Aritmetické metametódy sa volajú pri nedefinovaných aritmetických operáciach. Poznáme niekoľko aritmetických metametód:

* \_\_add – operácia sčítania (napr. *a* + *b*, kde aspoň jeden operand nepredstavuje číselnú hodnotu). Túto metametódu možno definovať pomocou (anonymnej) funkcie, ktorá prijíma dva operandy ako argumenty (*a* a *b*). Výsledok sa vracia pomocou príkazu return, ktorý sa považuje za výsledok operácie.
* \_\_sub – operácia odčítania (napr. *a* - *b*). Správanie je podobné ako pri metametóde \_\_add.
* \_\_mul – operácia násobenia (napr. *a* \* *b*). Správanie je podobné ako pri metametóde \_\_add.
* \_\_div – operácia delenia (napr. *a* / *b*). Správanie je podobné ako pri metametóde \_\_add.
* \_\_mod – operácia modulo (napr. *a* % *b*). Správanie je podobné ako pri metametóde \_\_add.
* \_\_pow – operácia umocnenia (napr. *a* ^ *b*). Správanie je podobné ako pri metametóde \_\_add.
* \_\_idiv – operácia *floor* delenia (napr. *a* // *b*). Správanie je podobné ako pri metametóde \_\_add.
* \_\_unm – operácia unárneho mínus (napr. -*a*, kde a nepredstavuje číselnú hodnotu). Správanie je podobné ako pri metametóde \_\_add s tým rozdielom, že (anonymná) funkcia prijíma iba jeden operand ako argument.

### 3.6.2 Bitové metametódy

Bitové metametódy sa volajú pri nedefinovaných bitových operáciach. Poznáme niekoľko bitových metametód:

* \_\_band – operácia bitový AND (napr. *a* & *b*). Správanie je podobné ako pri metametóde \_\_add.
* \_\_bor – operácia bitový OR (napr. *a* | *b*). Správanie je podobné ako pri metametóde \_\_band.
* \_\_bxor – operácia bitový XOR (napr. *a* ~ *b*). Správanie je podobné ako pri metametóde \_\_band.
* \_\_bnot – operácia bitový NOT (napr. ~*a*, kde a nepredstavuje číselnú hodnotu). Správanie je podobné ako pri metametóde \_\_band s tým rozdielom, že (anonymná) funkcia prijíma iba jeden operand ako argument.
* \_\_shl – operácia bitového posunutia vľavo (napr. *a* << *b*). Správanie je podobné ako pri metametóde \_\_band.
* \_\_shr – operácia bitového posunutia vpravo (napr. *a* >> *b*). Správanie je podobné ako pri metametóde \_\_band.

### 3.6.3 Porovnávacie metametódy

Porovnávacie metametódy sa volajú pri nedefinovaných porovnávacích operáciach. Poznáme niekoľko porovnávacích metametód:

* \_\_eq – operácia *sa rovná* (napr. *a* == *b*, kde oba operandy sú buď typu table alebo userdata, pričom sa primitívne nerovnajú). Správanie je podobné ako pri metametóde \_\_add s tým rozdielom, že výsledok vrátený pomocou príkazu return je konvertovaný na *boolean* hodnotu.
* \_\_lt – operácia *je menej ako* (napr. *a* < *b*, kde oba operandy nie sú ani číselné hodnoty, ani reťazce). Správanie je podobné ako pri metametóde \_\_eq.
* \_\_le – operácia *je menej ako alebo sa rovná* (napr. *a* <= *b*). Správanie je podobné ako pri metametóde \_\_lt.

### 3.6.4 Tabuľkové metametódy

Tabuľkové metametódy sa volajú pri nedefinovaných operáciach nad tabuľkou. Poznáme niekoľko tabuľkových metametód:

* \_\_index – indexový prístup k hodnote (napr. *t*[*k*]). Táto metametóda sa použije v prípade, že *t* nie je tabuľka, alebo *t* neobsahuje kľúč *k*. Túto metametódu možno definovať pomocou (anonymnej) funkcie, ktorá prijíma dva argumenty. Ako prvý prijíma hodnotu *t*, a ako druhý prijíma hodnotu *k*. Výslednú hodnotu táto funkcia vracia pomocou príkazu return. Druhou možnosťou, ako definovať túto metametódu je, že namiesto (anonymnej) funkcie jej priradíme tabuľku, ku ktorej sa bude pristupovať pomocou kľúča *k*.
* \_\_newindex – indexové priradenie hodnoty (napr. *t*[*k*] = *v*). Táto metametóda sa použije v prípade, že *t* nie je tabuľka, alebo *t* neobsahuje kľúč *k*. Túto metametódu možno definovať pomocou (anonymnej) funkcie, ktorá prijíma tri argumenty. Ako prvý prijíma hodnotu *t*, ako druhý prijíma hodnotu *k*, a ako tretí prijíma hodnotu *v*. Druhou možnosťou, ako definovať túto metametódu je, že namiesto (anonymnej) funkcie jej priradíme tabuľku, s ktorou sa bude manipulovať pomocou kľúča *k* a hodnoty *v*.

### 3.6.5 Ostatné metametódy

Medzi ostatné metametódy patria:

* \_\_metatable – tejto metametóde možno priradiť hocijakú hodnotu (okrem nil). Tým sa zakáže zmena metatabuľky pomocou funkcie setmetatable objektu, ktorý túto metatabuľku vlastní. Ak sa pokúsime o zmenu metatabuľky, tak sa vyvolá chyba. Jedná sa o ochranný mechanizmus. Ak zavoláme funkciu getmetatable, tak dostaneme presne túto hodnotu.
* \_\_concat – operácia spojenia (napr. *a* .. *b*, kde aspoň jeden operand nepredstavuje ani reťazec, ani číselnú hodnotu). Správanie je podobné ako pri metametóde \_\_add.
* \_\_len – operácia dĺžky (napr. #*a*, kde *a* nepredstavuje reťazec). Túto metametódu možno definovať pomocou (anonymnej) funkcie, ktorá prijíma argument *a*. Výslednú dĺžku vracia pomocou príkazu return, ktorá sa považuje za výsledok operácie. V prípade, že argument *a* je tabuľka, a túto metametódu nemá definovanú, tak sa ako výsledok vráti „skutočná“ dĺžka tabuľky *a*.
* \_\_call – operácia volania (napr. *a*(*args*), kde *a* nepredstavuje funkciu). Túto metametódu možno definovať pomocou (anonymnej) funkcie, ktorá prijíma, za prvým argumentom *a*, aj extra argumenty *args*. Výsledné hodnoty vracia pomocou príkazu return, ktoré sa považujú za výsledok operácie.
* \_\_tostring – operácia *prevod na reťazec* (napr. print(*a*)). Túto metametódu možno definovať pomocou (anonymnej) funkcie, ktorá vracia reťazec reprezentujúci argument *a*v textovej podobe.

### Vzorový príklad pr3\_6

Vzorová aplikácia názorne demonštruje jednotlivé metametódy v akcii. Nenájdeme v nej však úplne všetky, ktoré sme vymenovali, nakoľko si je väčšina z nich podobná. Nájdeme v nej aj výpis, ktoré typy hodnôt majú metatabuľky štandardne nastavené. Za povšimnutie stojí metametóda \_\_eq, pretože pri nej vzniká problém. Okrem iného používame v našej aplikácii funkcie setmetatable, getmetatable, rawset, rawget, rawequal, rawlen.

## 3.7 Objektovo orientované programovanie

Existuje veľa programovacích jazykov, ktoré nám umožňujú využívať programovaciu paradigmu objektovo orientovaného programovania. Programovací jazyk Lua túto programovaciu paradigmu natívne nepodporuje, avšak existuje spôsob, ako ju doňho zaviesť. Tým spôsobom sú tabuľky. Pomocou nich možno napodobniť koncept tried.

Tabuľka v programovacom jazyku Lua je objektom vo viac ako len jednom význame. Tak isto ako objekty, tak aj tabuľky majú stav. Tak isto ako objekty, tak aj tabuľky majú identitu, ktorá je nezávislá od jej hodnôt. To znamená, že dva objekty, resp. tabuľky, ktoré majú rovnaké hodnoty sú odlišnými objektami, pričom každý objekt môže mať rozdielne hodnoty v rozdielnom čase. Tak isto ako objekty, tak aj tabuľky majú životný cyklus, ktorý je nezávislý od toho, kto ich vytvoril alebo kde boli vytvorené (Ierusalimschy, 2013).

### 3.7.1 Návrh triedy

Spôsobov, ako navrhnúť triedu existuje viacero. My opíšeme len jeden. Každá trieda by mala byť uložená v samostatnom súbore. Na začiatku súboru deklarujeme lokálnu premennú *c* s názvom danej triedy. Do tejto premennej priradíme prázdnu tabuľku. Táto tabuľka predstavuje prototyp triedy. Na nasledujúcich riadkoch definujeme atribúty s predvolenými hodnotami. Názvy týchto atribútov prefixujeme pomocou lokálnej premennej *c* a bodky (jedná sa o klasické zapísanie hodnoty do tabuľky), napr.: App.isRunning = false. Pokiaľ nejaký atribút nemá predvolenú hodnotu, resp. jeho predvolená hodnota je nil, tak ho síce môžeme definovať, ale je to zbytočné. K tomuto môže napríklad dôjsť, keď chceme daný atribút okomentovať. Tabuľky nám neumožňujú označiť jednotlivé kľúče ako súkromné (*private*) alebo chránené (*protected*). Aby sme dali najavo, že daný atribút je súkromný, a nemalo by sa k nemu pristupovať zvonku, tak ho môžeme prefixovať pomocou znaku \_. Po atribútoch definujeme konštruktor. Na názve konštruktora nezáleží, štandardne mu ale môžeme dať názov new. Príkladná definícia konštruktora môže vyzerať nasledovne:

function App.new(prototype, title, version) ... end

Ako si môžeme všimnúť, tak konštruktor je obyčajná funkcia. Za povšimnutie stojí aj prvý parameter prototype. Tento parameter bude ako argument prijímať tabuľku reprezentujúcu prototyp triedy. V danom príklade nesie prototyp triedy lokálna premenná App. Nový objekt typu App vytvoríme nasledovne:

local myapp = App:new(“Test“, “1.0“)

Ako si môžeme všimnúť, tak konštruktoru sme odovzdali iba dva argumenty, a to: názov a verziu. Argument prototype sa odovzdal automaticky vďaka dvojbodke medzi App a new. Argument prototype môžeme odovzdať aj manuálne, pokiaľ medzi App a new napíšeme bodku namiesto dvojbodky, teda: App.new(App, “Test“, “1.0“). Avšak, preferovaný spôsob je s dvojbodkou. Dôvod, prečo v deklarácii konštruktora používame parameter prototype, je dedičnosť. Ak plánujeme od triedy odvodiť inú triedu, tak tento parameter budeme potrebovať. V opačnom prípade môžeme tento parameter vynechať. V tom prípade, by sme nový objekt typu App vytvorili nasledovne:

local myapp = App.new(“Test“, “1.0“)

Konštruktor ako taký musí vytvoriť novú tabuľku, ktorá bude reprezentovať samotný objekt (inštanciu danej triedy), ktorý vráti pomocou príkazu return. Do tejto tabuľky pridáme všetky atribúty, ktoré nemajú definovanú predvolenú hodnotu, alebo ktorých hodnoty sa líšia od tých predvolených. Následne tejto tabuľke nastavíme ako metatabuľku argument prototype, pričom do tabuľky v prototype pridáme metametódu \_\_index:

setmetatable(object, prototype)

prototype.\_\_index = prototype

V prípade, že daný objekt neobsahuje požadovaný atribút alebo operáciu, tak sa to pokúsi nájsť v prototype triedy. To je úloha tých dvoch riadkov vyššie. Po konštruktore, na nasledujúcich riadkoch, definujeme všetky operácie, ktoré potrebujeme. Príkladná definícia operácie môže vyzerať nasledovne:

function App:run(asDaemon) ... end

Na rozdiel od konštruktora sme medzi App a run použili dvojbodku namiesto bodky. Význam tejto dvojbodky je, že pridáva skrytý parameter self pred prvý parameter (v našom prípade asDaemon). Tento spôsob zápisu je preferovaný. Parameter self môžeme deklarovať aj vlastnoručne, napr. keď chceme, aby sa volal this namiesto self. V tomto prípade by sme medzi App a run napísali bodku. Príkladná definícia operácie by vyzerala nasledovne:

function App.run(this, asDaemon) ... end

Je jedno, akým spôsobom operácie deklarujeme, a je jedno, akým spôsobom ich zavoláme. Ak sme operáciu deklarovali pomocou dvojbodky, tak ju môžeme zavolať pomocou dvojbodky, ako aj pomocou bodky. To isté platí pre deklaráciu pomocou bodky analogicky. Príkladné volania na objekte myapp môžu vyzerať nasledovne:

myapp:run(true)

myapp.run(myapp, true)

Oba zápisy sú totožné. Keďže trieda je definovaná pomocou prototypu, tak samotné objekty si môžu jednotlivé operácie prepísať. Príkladný prepis operácie prototypu môže vyzerať nasledovne:

myapp.run = function (self, asDaemon) ... end

Na záver môžeme samotnému prototypu triedy nastaviť metatabuľku s metametódou \_\_call, ktorá sa bude správať presne ako konštruktor. Ak toto vylepšenie aplikujeme na prototyp triedy App, tak nám to umožní vytvoriť nový objekt aplikácie prostredníctvom App(“Test“, “1.0“) namiesto dlhšieho zápisu App:new(“Test“, “1.0“). Ak sme hotoví, tak na konci súboru vrátime tabuľku reprezentujúcu prototyp triedy pomocou príkazu return.

### 3.7.2 Návrh odvodenej triedy

Implementovať triedu, ktorú rozširuje iná trieda nie je v programovacom jazyku Lua nič zložitého. Najskôr definujeme prototyp triedy tak, ako sme to uviedli v predchádzajúcej podkapitole. Prototypu triedy nastavíme metatabuľku s metametódou \_\_index, ktorá nenájdené kľúče bude hľadať v rodičovskej triede. Tomuto prototypu triedy musíme upraviť aj konštruktor. V konštruktore vytvoríme nový objekt rodičovskej triedy, pričom jej konštruktor voláme cez bodku. Ako prvý argument uvedieme prototyp odvodenej triedy. Ak máme k dispozícii argument prototype, tak odovzdáme ten. Následne do tohto objektu môžeme pridať ďalšie atribúty alebo prepísať tie jestvujúce. Nakoniec ho vrátime pomocou príkazu return. V odvodenej triede môžeme definovať nové metódy alebo prepísať tie z rodičovskej triedy.

### 3.7.3 Návrh triedy so súkromnými atribútmi

Programovací jazyk Lua neposkytuje žiadny natívny mechanizmus obmedzenia prístupu k dátam tabuľky. Všetky dáta tabuľky sú teda verejne dostupné. Napriek tomu, programovací jazyk Lua je tak flexibilný, že umožňuje napodobniť niektoré z existujúcich mechanizmov. Avšak, tieto *napodobneniny* sa príliš často nepoužívajú a radšej sa všetci držia hesla: „Ak k niektorým dátam tabuľky pristupovať nechceš, tak to jednoducho nerob.“

Ako prvé definujeme lokálnu funkciu, ktorej názov štandardne začína slovom new (napr. newClient). Táto funkcia predstavuje konštruktor. V rámci tejto funkcie definujeme lokálne premenné, ktoré predstavujú samotné atribúty. Definujeme úplne všetky, bez ohľadu na to, či majú predvolenú hodnotu alebo nie. Následne pomocou príkazu return vrátime novú tabuľku, ktorá obsahuje definície jednotlivých operácií. Tieto operácie pristupujú a menia jednotlivé atribúty (teda lokálne premenné). Danú funkciu na konci súboru vrátime pomocou príkazu return. To, čo nám daná funkcia po jej zavolaní vráti je len tabuľka s operáciami, ktorá predstavuje samotný objekt, pričom dáta sú dostupné a meniteľné iba prostredníctvom daných operácií.

### Vzorový príklad pr3\_7

Uvedené koncepty a postupy demonštrujeme v rámci našej vzorovej aplikácie simulujúcej bakomat. Aplikácia si po spustení vypýta číslo bankomatovej karty a PIN kód. K dispozícii je napríklad 101010 a 1010. Ďalšie páry možno nájsť na vrchu súboru atm.lua. Ďalšie inštrukcie poskytuje samotná aplikácia.

Tabuľka Zoznam súborov vzorového príkladu pr3\_7

|  |  |
| --- | --- |
| **Súbor** | **Popis** |
| main.lua | spúšťací súbor programu |
| atm.lua | obsahuje definíciu bankomatu |
| acc.lua | obsahuje definíciu rodičovskej triedy účtu (*3.7.1*) |
| credit\_acc.lua | obsahuje definíciu triedy kreditného účtu (*3.7.2*) |
| debit\_acc.lua | obsahuje definíciu triedy debetného účtu (*3.7.2*) |
| card.lua | obsahuje definíciu triedy bankomatovej karty (*3.7.1*) |
| client.lua | obsahuje definíciu triedy klienta (*3.7.3*) |

## 3.8 Moduly

Za modul možno označiť súbor, ktorý v sebe definuje určitý počet funkcií a premenných, a ktorý všetky tieto funkcie a premenné alebo len časť z nich exportuje, aby sa dali použiť aj v inom súbore. Prvou možnosťou ako vytvoriť modul je, že na začiatku súboru si vytvoríme prázdnu tabuľku, postupne ako v súbore definujeme nové funkcie a premenné, tak ich do tejto tabuľky pridávame, a na konci súboru túto tabuľku vrátime pomocou príkazu return. Táto tabuľka sa správa ako menný priestor (*namespace*). Do tejto tabuľky, samozrejme, nemusíme pridať úplne všetky funkcie a premenné, ktoré sme v danom súbore definovali, ale len tie, ktoré chceme sprístupniť zvonku. Tie funkcie a premenné, ktoré majú zostať interné, treba označiť kľúčovým slovom local, inak by sme k nim mohli pristupovať zvonku, pretože by boli definované ako globálne. Názvy funkcií a premenných musíme prefixovať premennou, ktorá reprezentuje menný priestor, a bodkou. Druhou možnosťou ako vytvoriť modul je, že v súbore označíme všetky funkcie a premenné kľúčovým slovom local, a na konci súboru vytvoríme tabuľku, ktorú naplníme jednotlivými funkciami a premennými. Túto tabuľku následne vrátime pomocou príkazu return. Vďaka tomuto riešeniu nemusíme prefixovať názvy funkcií a premenných tak, ako pri prvej možnosti. Treťou možnosťou je, že modul definujeme v rámci inicializačnej funkcii, ktorú vrátime príkazom return. Inicializačná funkcia môže sama o sebe prijímať inicializačné argumenty. Tento princíp je podobný tomu, ktorý sme popísali v podkapitole *3.7.3 Návrh triedy so súkromnými atribútmi*. Samotný modul potom môžeme, v rámci inicializačnej funkcii, definovať druhým spôsobom. Využitie by sme napríklad mohli nájsť pri module, ktorý by poskytoval funkcie na meranie vzdialeností. Inicializačnej funkcii by sme odovzdali typ jednotky, v ktorých majú funkcie merať (napr. v centimetroch, metroch, ...). Vďaka tomuto riešeniu nemusíme každej jednej funkcii odovzdávať typ jednotky, ale len raz ho určíme pri inicializácii. V rámci jedného súboru môžeme mať načítané aj dva takéto moduly, ktoré sa odlišujú iba typom jednotky, čo možno považovať tiež za výhodu. Medzi nevýhody zasa patria pamäťové nároky, pretože vždy keď zavoláme inicializačnú funkciu, tak nám interne vytvorí *nový* modul. Záleží to ale aj od toho, ako samotný modul navrhneme. K tejto nevýhode nemusí takmer vôbec dôjsť. Okrem týchto spôsobov sa môžeme dopátrať aj k ďalším spôsobom ako modul definovať. Napriek tomu, je jedno ktorý spôsob si vyberieme, dôležité je, aby používatelia modulu boli schopní ho použiť štandardnou cestou:

local mojmodul = require(“mojmodul“)

-- ďalej už pracujeme iba s premennou mojmodul

Názvy modulov môžu obsahovať aj znak \_, ktorý môže slúžiť ako oddeľovač slov. Možno použiť aj znak – ak chceme verzionovať súbory. S týmto znakom však musíme byť opatrní, pretože programovací jazyk Lua definuje určité pravidlá, za akých modul zo súboru načítava. Lepšie je sa tomuto znaku radšej vyhnúť.

### 3.8.1 Submoduly

Programovací jazyk Lua umožňuje hierarchicky usporadúvať moduly. Hierarchické usporiadanie si môžeme predstaviť na príklade, kedy máme moduly mojmodul a mojmodul.mojpomocnymodul. Modul mojmodul.mojpomocnymodul je v tomto prípade submodulom modulu mojmodul. Submoduly sa oddeľujú bodkou. Submodulov môžeme mať v rade za sebou aj niekoľko. To znamená, že aj submodul môže mať ďalšie submoduly. Submoduly spolu s *hlavným* modulom vytvárajú tzv. balíček. Každý submodul je definovaný vo vlastnom súbore. Preto treba aj tieto súbory v rámci projektu hierarchicky usporiadať do adresárovej štruktúry. Adresárová štruktúra modulu mojmodul by mohla vyzerať nasledovne:

* ~/mojmodul.lua alebo ~/mojmodul/init.lua
* ~/mojmodul/mojpomocnymodul.lua alebo ~/mojmodul/mojpomocnymodul/init.lua

Ako môžeme vidieť, tak bodka medzi modulom a submodulom sa zmenila na znak adresárového oddeľovača, plus sa pridala prípona súboru .lua. Ako si môžeme ďalej všimnúť, tak (sub)modul možno definovať aj v súbore init.lua. Teoreticky to môže byť aj súbor s iným názvom, štandardne sa však používa práve init.lua. Závisí to ale od toho, ako máme definovanú cestu k súboru, resp. jej vzor, na ktorej sa má daný (sub)modul hľadať.

### 3.8.2 Načítanie modulu

Modul, resp. submodul, načítame pomocou funkcie require. Táto funkcia prijíma ako argument názov modulu (napr. require(“mojmodul“)). Pokiaľ chceme načítať submodul, tak jednotlivé názvy modulov oddelíme bodkou (napr. require(“mojmodul.mojpomocnymodul“)). Bodka sa v tomto prípade mení na adresárový oddeľovač. Možné je oddeliť jednotlivé názvy modulov aj lomítkom. Ak vzor cesty k súboru vyzerá nasledovne: ~/program/?.lua;~/program/?/init.lua, a ak sa pokúsime pomocou funkcie require načítať submodul mojmodul.mojpomocnymodul, tak sa vlastne pokúšame načítať jeden z týchto súborov: ~/program/mojmodul/mojpomocnymodul.lua alebo ~/program/mojmodul/mojpomocnymodul/init.lua. Znak ? sa nahrádza za názov modulu. Ak ani jeden z týchto súborov neexistuje, tak dôjde k chybe. Ak existujú oba súbory, tak sa načíta ten prvý, pretože hľadanie súboru sa vykonáva zľava doprava. Funkcia require vracia to, čo v danom súbore, na jeho konci, je priradené k príkazu return. V prípade tabuľky s definíciami danú tabuľku, v prípade inicializačnej funkcii danú funkciu. Preto, to čo funkcia require vráti väčšinou priraďujeme do lokálnej premennej. Ak by sme v rámci jedného súboru zavolali dva a viackrát funkciu require s tým istým modulom, tak vždy by nám vrátila ten istý objekt. Je to preto, lebo funkcia require si *cache-uje* výsledky do tabuľky uloženej v poli package.loaded, kde ako kľúč je použitý názov (sub)modulu.

### Vzorový príklad pr3\_8

V rámci nášho vzorového príkladu demonštrujeme všetko to, čo sme tu rozobrali. Definujeme tu dva moduly, plus a star, ktoré sú funkcionálne totožné. Rozdiel je iba v znaku, ktorý ich reprezentuje. Tým sme chceli ukázať, že vďaka menným priestorom môžeme definovať funkcie a premenné s rovnakými názvami, pričom nám dávajú rozdielne výsledky. Definujeme tu aj modul calculator, ktorý môžeme vytvoriť viackrát pomocou inicializačnej funkcie.

Tabuľka Zoznam súborov vzorového príkladu pr3\_8

|  |  |
| --- | --- |
| **Súbor** | **Popis** |
| main.lua | spúšťací súbor programu |
| plus.lua | obsahuje definíciu modulu plus |
| star.lua | obsahuje definíciu modulu star |
| calculator/init.lua | obsahuje definíciu modulu calculator |

## 3.9 Vloženie do hostiteľskej aplikácie

V programovacom jazyku Lua len veľmi ťažko vytvoríme plnohodnotnú aplikáciu. Preto sa vo väčšine prípadov vkladá do hostiteľskej aplikácie. Vloženie do hostiteľskej aplikácie využívame práve vtedy, keď je oveľa jednoduchšie vyriešiť daný problém v inom, jednoduchšom programovacom jazyku (v našom prípade v programovacom jazyku Lua) ako v tom primárnom. Môžeme ho využiť aj vtedy, ak chceme klientom poskytnúť nejakú formu verejného API, pomocou ktorého môžu aj oni rozširovať danú aplikáciu. K tomuto dochádza najmä vtedy, keď očakávame menšie znalosti v programovaní u klienta. V praxi to znamená, že ak programujeme napríklad v programovacom jazyku C, tak pre veľa ľudí môže byť tento programovací jazyk príliš náročný. Aby sme im to uľahčili, poskytneme im možnosť programovať v programovacom jazyku Lua, ktorý je oveľa jednoduchší a ľahšie naučiteľný. Programovací jazyk Lua môžeme interpretovať cez programovacie jazyky C, C++, Java, PHP, JavaScript, Ruby a mnoho ďalších. My popíšeme vloženie len cez tri z nich. A na záver predstavíme dva vzorové príklady, ktoré sa nachádzajú v priečinku pr3\_9.

### 3.9.1 Vloženie cez programovací jazyk C

Samotný programovací jazyk Lua je napísaný v programovacom jazyku C. Preto jeho interpretácia cezeň nie je žiadny problém. Lua API v programovacom jazyku C nám poskytuje veľa funkcií a konštánt, ktoré začínajú prefixom lua\_, resp. LUA\_. Toto API nám toho dovoľuje, prirodzene, najviac. Lua API však nie je súčasťou C API.

### 3.9.2 Vloženie cez programovací jazyk Java

Programovací jazyk Java natívne nepodporuje interpretáciu programovacieho jazyka Lua. Avšak, k dispozícii máme knižnicu *LuaJ*, ktorá túto podporu pridáva. Táto knižnica nám poskytuje niekoľko tried, napríklad JsePlatform, Globals, LuaValue, LuaTable a pod. Pre znázornenie, pomocou príkazu Globals globals = JsePlatform.standardGlobals(); si do premennej globals vieme načítať štandardné globálne premenné programovacieho jazyka Lua (napr. require, print, a pod.) Následne na to ich vieme z premennej globals aj volať.

### 3.9.3 Vloženie cez programovací jazyk PHP

Programovací jazyk PHP natívne nepodporuje interpretáciu programovacieho jazyka Lua. Avšak, túto podporu môžeme pridať prostredníctvom PECL rozšírenia *Lua*. Toto rozšírenie nám poskytuje triedu pomenovanú Lua. Táto trieda nám poskytuje metódu call na zavolanie funkcie, metódu assign na priradenie hodnoty do premennej, metódu eval na vykonanie príkazov a pod. Pokiaľ potrebujeme zistiť akú verziu programovacieho jazyka Lua intrepretuje programovací jazyk PHP, k dispozícii máme metódu getVersion.

### Vzorový príklad pr3\_9/php: PHP aplikácia volajúca Lua skript

Touto aplikáciou chceme názorne demonštrovať spôsob, akým si môžeme vytvoriť vlastnú aplikáciu, ktorá bude volať skripty napísané v programovacom jazyku Lua. Jedná sa o jednoduchú PHP aplikáciu, ktorej keď zadáme určitý zoznam hodnôt, tak po spracovaní nám vráti zoznam jedinečných hodnôt usporiadaný abecedne. Obrázok tejto aplikácie sa nachádza v prílohe B.1. V rámci súboru script.lua definujeme globálne funkcie string\_to\_list, list\_to\_set a order, ktoré voláme priamo v súbore index.php. Súbor script.lua však musíme v súbore index.php najskôr načítať pomocou triedy Lua. Ak by sme chceli zmeniť správanie týchto funkcií, tak by sme jednoducho mohli vytvoriť napríklad súbor script2.lua, a ten následne načítať v súbore index.php namiesto súboru script.lua. V súbore script2.lua by sme samozrejme tieto funkcie prepísali.

### Vzorový príklad pr3\_9/fxcm: Indikátor predaja a nákupu

Pokiaľ nevyvíjame vlastnú aplikáciu, ale naopak, sme používateľom nejakej už existujúcej, tak sa môže stať, že nám bude poskytovať verejné API dostupné práve v programovacom jazyku Lua. Takýmto príkladom je napríklad obchodná platforma *FXCM Trading Station*. Tá nám poskytuje možnosť vytvoriť si vlastné indikátory a stratégie práve cez tento jazyk. My sme jeden vzorový indikátor naprogramovali. V prílohe B.2 môžete vidieť obrázok cenového grafu bez nášho indikátora. Samotné API nám prikazuje vytvoriť globálne funkcie Init, Prepare a Update. Vo funkcii Init definujeme všetky informácie o indikátore, napr. názov, popis, voliteľné parametre a pod. Obrázok v prílohe B.3 nám ukazuje tieto informácie v praxi. Dané okno nám vyskočí, keď sa pokúšame pridať indikátor na cenový graf. Vo funkcii Prepare inicializujeme samotnú inštanciu indikátora. Vo funkcii Update počítame výstupné hodnoty indikátora. Táto funkcia sa volá pri každej aktualizácii ceny. Samotný výstup indikátora môžeme vidieť na obrázku v prílohe B.4. Došlo iba k farebnému prekrytiu *sviečok*. Úloha tohto indikátora spočíva v tom, že keď máme zelené *sviečky*, tak máme nakupovať obchodný inštrument. Keď červené, tak ho predávať.

# Záver

Detailne popísať akýkoľvek programovací jazyk na pár stranách je takmer nemožné. My sme sa však pokúsili jeden z nich popísať aspoň tak čiastočne. Keďže pohľad z teoretickej časti v praxi nestačí, čitateľovi sme sa snažili poskytnúť pohľad aj z tej praktickej časti.

Cieľom bakalárskej práce bolo preskúmať a popísať vlastnosti, koncepty a postupy typické pre programovanie v programovacom jazyku Lua. Na základe získaných vedomostí a vlastných praktických skúseností sme vytvorili kolekciu vzorových aplikácií, ktoré názorne demonštrujú použitie programovacieho jazyka Lua v praxi. Veríme, že naša bakalárska práca môže pomôcť aj ďalším potenciálnym používateľom tohto jazyka, nakoľko je písaná v slovenskom jazyku.

V bakalárskej práci sme sa zamerali, z nášho pohľadu, na tie najhlavnejšie časti programovacieho jazyka Lua ako celkové základy, dátové štruktúry, funkcie, korutiny, chyby, metatabuľky a metametódy, objektovo orientované programovanie, moduly a samotné vloženie do hostiteľskej aplikácie. Kvôli rozsahu bakalárskej práce sme sa nevenovali aj ďalším dôležitým častiam ako práca so súbormi, vstavané moduly, správa pamäte, iterátory, kompilácia, prostredie, memoizácia, Lua API v jazyku C a pod.

Po oboznámení sa s programovacím jazykom Lua môžeme konštatovať, že svoje miesto si medzi širokým sortimentom programovacích jazykov určite zaslúži. Z nášho pohľadu to nie je len ďalší jazyk navyše. Svedčí o tom aj jeho veľké uplatnenie v hernom priemysle. Ak raz budeme tvoriť aplikáciu a našim klientom budeme chcieť poskytnúť verejné API, určite budeme zvažovať nasadenie tohto jazyka. Najmä kvôli jeho jednoduchosti a rýchlosti. Bolo by to na spôsob, ktorý sme mohli vidieť u obchodnej platformy *FXCM Trading Station*.

Pre potenciálnych záujemcov o programovanie v jazyku Lua môžeme, z vlastnej skúsenosti, odporučiť knihu *Programming in Lua*. Kvalitné online zdroje sme uviedli aj v analýze súčasného stavu. Pokiaľ chceme programovací jazyk Lua zatiaľ len vyskúšať bez inštalácie, k dispozícii máme online textový editor (vrátane zvýrazňovania syntaxe) spolu s interpreterom dostupným na webovej stránke *www.tutorialspoint.com/execute\_lua\_online.php*.

# Zoznam bibliografických odkazov

DAVIS, J. 2013. The Differences Between Imperative and Functional Programming. Part 1. [online]. [cit. 2015-11-10]. Dostupné na internete: <https://joshldavis.com/2013/09/30/difference-between-imperative-and-functional-part-1/>.

ETANDEL. 2012. Lua, what is Lua? [online]. [cit. 2015-11-13]. Dostupné na internete: <http://stackoverflow.com/questions/1274972/lua-what-is-lua>.

IERUSALIMSCHY, R. 2013. Programming in Lua, Third Edition.366 s. ISBN 978-8590379850.

JUNG, K. A BROWN, A. 2007. Beginning Lua Programming.672 s. ISBN 978-0470069172.

KAŠUBA, M. 2015. Lua Game Development Cookbook.402 s. ISBN 978-1849515504.

LEWIS, J. A LOFTUS, W. 2015. Java Software Solutions: Foundations of Program Design (8th Edition). 832 s. ISBN 978-0133594959.

KARPAT, M. 2007. Komunikácia medzi autami - vyskúšali sme technológiu budúcnosti. [online]. [cit. 2019-11-30]. Dostupné na internete: <https://auto.sme.sk/c/3469935/komunikacia-medzi-autami-vyskusali-sme-technologiu-buducnosti.html>.

LUA.ORG. 2015. Lua: about. [online]. [cit. 2015-11-14]. Dostupné na internete: <http://www.lua.org/about.html>.

LUA.ORG. 2016. Programming in Lua. [online]. [cit. 2016-03-20]. Dostupné na internete: <http://www.lua.org/pil/>.

LUAEDIT.SOURCEFORGE.NET. 2015. LuaEdit. [online]. [cit. 2015-11-21]. Dostupné na internete: <http://luaedit.sourceforge.net/>.

LUAROCKS.ORG. 2016. LuaFileSystem - LuaRocks. [online]. [cit. 2016-03-20]. Dostupné na internete: <https://luarocks.org/modules/hisham/luafilesystem>.

LUA-USERS.ORG. 2015. Lua Uses. [online]. [cit. 2015-11-16]. Dostupné na internete: <http://lua-users.org/wiki/LuaUses>.

MLADY-VEDEC.EU. 2015. História Programovania. [online]. [cit. 2016-03-18]. Dostupné na internete: <http://www.mlady-vedec.eu/blog/entry/historia-programovania>.

PEOPLE.CS.CLEMSON.EDU. 2015. Programming (Language) Paradigms. [online]. [cit. 2015-11-10]. Dostupné na internete: <http://people.cs.clemson.edu/~turner/courses/cs428/current/webct/content/pz/ch2/ch2\_6.html>.

ROGALSKY, T. 2012. My Favourite Lua Libraries. [online]. [cit. 2015-11-17]. Dostupné na internete: <http://tannerrogalsky.com/blog/2012/09/19/favourite-lua-libraries/>.

SKOUPIL, D. 2007. Úvod do paradigmat programování. [online]. [cit. 2015-11-10]. Dostupné na internete: <http://phoenix.inf.upol.cz/esf/ucebni/uvod\_para.pdf>.

W3.IMPA.BR. 2015. LuaSocket: Network support for the Lua language. [online]. [cit. 2015-11-18]. Dostupné na internete: <http://w3.impa.br/~diego/software/luasocket/>.

# Zoznam príloh

Príloha A – Vzorový zdrojový kód

Príloha B

* Príloha B.1 – PHP aplikácia volajúca Lua skript
* Príloha B.2 – Cenový graf bez indikátora
* Príloha B.3 – Voliteľné parametre indikátora
* Príloha B.4 – Cenový graf s indikátorom

Príloha C – CD nosič

* Príloha C.1 – Zdrojový kód vzorových aplikácií
* Príloha C.2 – Bakalárska práca vo formáte PDF

Príloha A

**local** mod = {}

-- číta nepárne riadky básničky zo súboru poem-odd-rows.txt

**local** **function** readOddRows(dirname, i)

**for** line **in** **io**.lines(dirname .. '/poem-odd-rows.txt') **do**

**local** dot = i < 10 **and** ". " **or** ". "

i = **coroutine**.yield(i .. dot .. line)

**end**

**return** " KONIEC SÚBORU poem-odd-rows.txt"

**end**

-- číta párne riadky básničky zo súboru poem-even-rows.txt

**local** **function** readEvenRows(dirname, i)

**for** line **in** **io**.lines(dirname .. '/poem-even-rows.txt') **do**

**local** dot = i < 10 **and** ". " **or** ". "

i = **coroutine**.yield(i .. dot .. line)

**end**

**return** " KONIEC SÚBORU poem-even-rows.txt"

**end**

**function** mod.readPoemFromTwoFiles(dirname)

**local** oddRowsCo = **coroutine**.create(readOddRows)

**local** evenRowsResume = **coroutine**.wrap(readEvenRows)

**local** i = 0

**local** isEvenRowsCoDead = **false**

**repeat**

**io**.read() -- ENTER

**if** **coroutine**.status(oddRowsCo) == 'suspended' **then**

**local** ok, str

i = i + 1

**if** i == 1 **then**

ok, str = **coroutine**.resume(oddRowsCo, dirname, i)

**else**

ok, str = **coroutine**.resume(oddRowsCo, i)

**end**

**if** ok **then** **print**(str) **else** **print**(" CHYBA: " .. str) **end**

**end**

**io**.read() -- ENTER;

**do**

i = i + 1

**local** ok = **pcall**(**function** ()

**if** i == 2 **then**

**print**(evenRowsResume(dirname, i))

**else**

**print**(evenRowsResume(i))

**end**

**end**)

isEvenRowsCoDead = **not** ok

**end**

**until** **coroutine**.status(oddRowsCo) == 'dead' **and** isEvenRowsCoDead

**end**

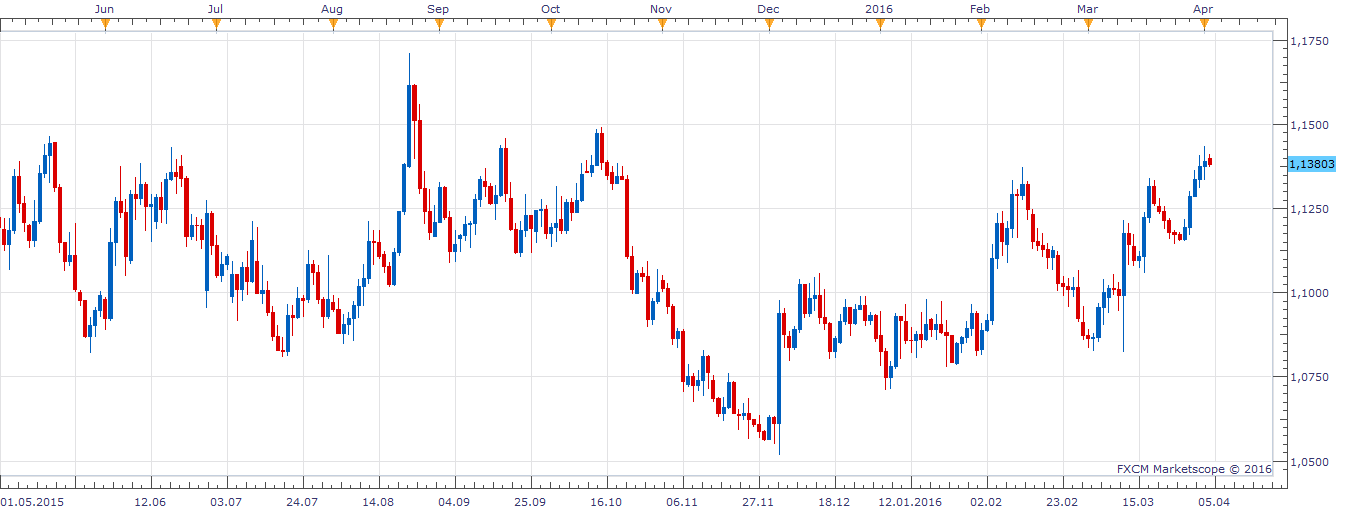
**return** mod

Príloha B

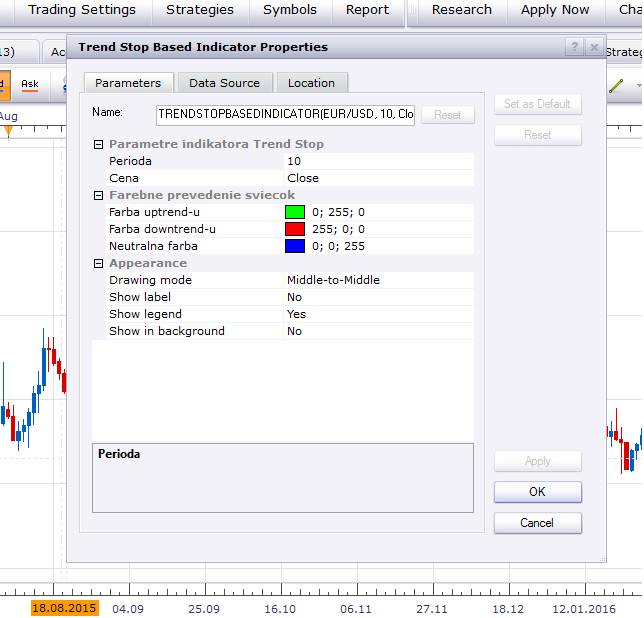
**Príloha B.1: PHP aplikácia volajúca Lua skript**



**Príloha B.2: Cenový graf bez indikátora**



**Príloha B.3: Voliteľné parametre indikátora**



**Príloha B.4: Cenový graf s indikátorom**

****

1. Index sa označuje aj ako kľúč, a opačne. [↑](#footnote-ref-1)