Programski jezik Lua

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Jovana Pejkić, Jana Jovičić, Katarina Rudinac, Ivana Jordanov
jov4ana@gmail.com, jana.jovicic755@gmail.com
rudinackatarina@gmail.com, ivanajordanov47@gmail.com

5. april 2019.

Sažetak

Pri rešavanju komplikovanih problema susrećemo se sa poteškoćom pri odabiru programskog jezika. Potrebno je da on bude fleksibilan, ali se ne bismo odrekli jednostavnosti ili dragocenog memorijskog prostora. Jedno od rešenja je programski jezik Lua. Cilj rada je pokazati kako je nastao ovaj programski jezik, kako se danas koristi i koje karakteristike poseduje. U te svrhe rad obuhvata opis njegovih fundamentalnih mehanizama, samim tim i načina kojim se složeni koncepti različitih programskih paradigmi posredstvom jednostavne semantike i svega 8 tipova podataka postižu u njemu.

Sadržaj

1 Uvod

Lua spada u dinamički tipizirane, gradivne (eng. embeddable[1]) skript jezike. Podržava proceduralno, objektno orjektisano, funkcionalno i programiranje vođeno podacima[2]. Nastao je kao potreba za fleksibilnim programskim jezikom koji će omogućiti jednostavno korišćenje nesrodnih mehanizama različitih programskih paradigmi, u jednoj zemlji u razvoju, a danas ima široku primenu u različitim oblastima programiranja. Koji su to mehanizmi i kako je nastala Lua biće opisano u poglavlju 2 , dok će o njenoj primeni danas biti reči u poglavlju 3. Načini na koji su podržane funckionalna i objektno orjentisana paradigma biće spomenuti u poglavlju 4. Kao rezultat širenja brojna su okruzenja za Luu i njima će biti posvećeno celo poglavlje 5. Jednostavnost programskog jezika će biti vidljiva kroz kodove koji demonstriraju ideju na kojoj je zasnovano programiranje u Lui (poglavlje 7), a pre toga u poglavlju 6 biće date instrukcije za instalaciju.

2 Nastanak

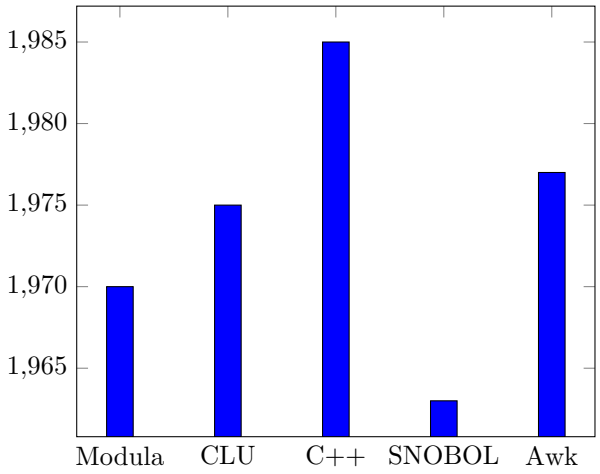
Programski jezik Lua nastao je 1993. na Katoličkom univerzitetu u Rio de Žaneiru u Brazilu, i u prevodu sa portugalskog znači mesec. U okviru Tecgraf-a, grupe za razvoj tehnologija u računarskoj grafici (eng. Computer Graphics Technology Group), pri pomenutom univerzitetu, trojica naučnika zvanih Roberto Jeruzalimski, Luiz Henrike de Figereido i Valdemar Keles, svaki specijalizovan za različitu naučnu oblast, razvili su jezik Lua, i od tada pa do danas rade na njegovom održavanju i proširivanju.

Zbog tadašnjih brazilskih zakona koji su striktno ograničavali uvoz softvera i hardvera, favorizujući domaće proizvode i znatno otežavajući uvoz iz inostranstva (koji je bio dozvoljen samo ukoliko ne postoji adekvatna zamena proizvedena od strane domaćih kompanija, što je bilo jako teško i komplikovano dokazati) stvorila se potreba za programskim jezikom nastalim u Brazilu. Lua se smatra jedinim programskim jezikom nastalim u nekoj zemlji u razvoju koji je dostigao globalnu popularnost, i pored Ruby nastalog u Japanu, jedinim značajnim jezikom nastalim van Evrope i severne Amerike.

Prethodnici Lue, dva mala programska jezika zvana DEL (Data Entry Language) i SOL (Simple Object Language, čija skraćenica u prevodu označava sunce, što je bila direktna inspiracija za naziv Lua), nastali su 1992. za potrebe brazilskog naftnog giganta Petrobrasa. Svrha naručenog grafičkog interfejsa je bila da olakša, ubrza i učini otpornijim na greške unos ogromne količine numeričkih podataka koji je morao biti unesen u zastarelom formatu nasleđenom iz doba bušenih kartica. Ovaj interfejs je učinio unos podataka interaktivnim, kliktanjem na delove dijagrama, a potom prevodio ovako unesene podatke u potrebni format i automatski generisao izlazni fajl u tom formatu. Pored toga, program je vršio proveru unetih podataka i izračunavao određeni deo podataka koji više nije bilo potrebe ručno unositi. DEL je nastao kako bi se olakšalo pisanje ovih grafičkih interfejsa, kao deklarativni jezik koji je opisivao svojstva

[1] embeddable language - klasa jezika koji se često koriste da prošire ili utiču na ponašanje postojeće aplikacije

[2] programiranje vodjeno podacima - data-driven programming is a programming paradigm in which the program statements describe the data to be matched and the processing required rather than defining a sequence of steps to be taken



Slika 1: Godine nastanka programskih jezika koji su uticali na Luu

i ograničenja polja za unos podataka i entiteta kojima su ta polja pripadala. Entiteti u DEL-u se mogu uporediti sa strukturama u nekom drugom jeziku. Na razvoju ovog jezika radio je Luiz Henrike de Figerido.

Paralelno sa razvojem DEL-a, razvijao se i PGM, na kojem su radili Roberto Jeruzalimski i Valdemar Keles. PGM je služio za ispis litoloških podataka i bilo je moguće ručno konfigurisati način ispisa. SOL je nastao kako bi se konfigurisanje olakšalo. SOL je završen marta 1993, ali nikada nije isporučen.

Ubrzo je nastala potreba za dodatnim mogućnostima u okviru DEL-a i SOL-a, tako da su vođe timova iza ova dva projekta sredinom 1993. zaključili da bi najbolje bilo da se ova dva mini jezika integrišu u jedan. Odlučeno je da će biti potreban pravi programski jezik, koji će da sadrži naredbe dodele, kontrole toka, podrutine... Zbog toga što ogroman deo potencijalnih korisnika nisu profesionalni programeri, sintaksa i semantika je trebalo da budu što jednostavnije, a postojala je i potreba za opisom podataka kao u SOL-u. Portabilnost je takođe označena kao jedan od prioriteta. DEL nije direktno uticao na Luu kao jezik, već samo idejno, u smislu umetanja delova skript jezika u ogromne projekte. Mnogi koncepti u Lui pozajmljeni su iz drugih programskih jezika, uključujući Modulu, CLU, C++, SNOBOL i Awk (prikaz hronologije nastanka ovih programskih jezika dat je na slici 1).

U julu 1993, Valdermar je dovršio prvu implementaciju Lue, koja je veoma brzo doživela ogroman uspeh. Prevedena je korišćenjem lex i yacc alata i bila je izuzetno jednostavna. Od tada je izašlo ukupno pet verzija sa brojnim podverzijama, koje su redom dodavale nove funkcionalnosti, a vremenski intervali između verzija su postajali sve duži.

3 Primena

Lua danas, zbog svoje jednostavnosti, efikasnosti i portabilnosti, ima primenu u najrazličitijim oblastima: ugradnim sistemima, mobilnim uređajima, veb serverima i igricama. Koristi se na jedan od sledeća tri načina: kao skript jezik u sastavu aplikacija pisanih na nekom drugom jeziku, kao

samostalan jezik ili zajedno sa C-om [7].

Ako se Lua upotrebljava u aplikaciji kao skript jezik, za njeno konfigurisanje potrebno je koristiti Lua-C API. Pomoću njega se mogu registrovati nove funkcije, praviti novi tipovi i vršiti izmene u ponašanju nekih operacija. Jedan od primera gde se Lua upotrebljava kao skript jezik je CGILua, alat za pravljenje dinamičkih veb stranica i manipulisanje podacima prikupljenih veb formama. U suštini, CGILua predstavlja apstrakciju za Veb server [9].

Kao skript jezik, Lua je našla primenu u razvoju softvera zasnovanom na komponentnom programiranju. U tom slučaju, Lua se koristi kao jezik za spajanje (eng. *glue language*) postojećih komponenti višeg nivoa - komponenti koje su napisane u kompajliranom, statički tipiziranom jeziku (kao što je C ili C++). One obično predstavljaju koncepte niskog nivoa koji se neće mnogo menjati tokom razvoja programa jer oduzimaju mnogo procesorskog vremena. Osim spajanja takvih komponenti, Lua se koristi za pisanje onih delova koda koji će verovatno biti menjani mnogo puta i na taj način ubrzava proces razvoja programa [7].

Veliki broj programera se sve češće opredeljuje za Luu prilikom pravljenja igrica, pre svega zbog njene efikasnosti i lake integracije sa drugim programskim jezicima. O popularnosti Lue u domenu igrica govori činjenica da je 2003. godine, na osnovu rezultata anketa sprovedenih na GameDev.net sajtu, proglašena za jezik koji je najbolji za pisanje skriptova za igrice [5].

Još jedan primer u kome se Lua koristi zajedno sa drugim programskim jezicima je Adobe Photoshop Lightroom. Ona omogućava pisanje dodatnih softverskih komponenti koje mogu da se integrišu sa postojećim programom. Na primer, može da služi za pisanje komponenti kojima se kreiraju različiti efekati, za dodavanje stavki u menije i dijaloge, manipulaciju metapodacima i kreiranje novih tipova veb galerija [1].

Sve češće, Lua se koristi i kao samostalan jezik, čak i za veće projekte. U te svrhe su razvijene biblioteke koje nude različite funkcionalnosti. Na primer, postoje biblioteke za rad sa stringovima, tabelama, fajlovima, modulima, korutinama, matematičkim funkcijama itd. Sve one zajedno formiraju standardnu biblioteku Lue.

Treća mogućnost za upotrebu Lue je u okviru programa pisanih u C-u. Tada se ona importuje kao C biblioteka. U praksi, najveći deo programa je u C-u, a Lua se koristi za kreiranje interfejsa lakih za upotrebu.

4 Podržane paradigme

Lua podržava različite paradigme, kao što su objektno-orjentisana, funkcionalna i proceduralna. Ona ne podržava ove paradigme pomoću specfičnih mehanizama za svaku od njih, već pomoću opštih mehanizama kao što su tabele, funkcije prvog reda, delegacija i korutine. Pošto ti mehanizmi nisu specifični ni za jednu određenu paradigmu, moguće su i druge paradigme. Svi mehanizmi Lue rade nad standardnom proceduralnom semantikom, što omogućuje njihovu laku integraciju. Prema tome, većina programa napisanih u Lui su proceduralni, s tim što uključuju i korisne tehnike drugih paradigmi [8] .

Podršku funkcionalnom programiranju u Lui pruža biblioteka Lua Fun, koja se još uvek razvija. Ona omogućava pisanje jednostavnog i efikasnog funkcionalnog koda korišćenjem funkcija višeg reda, poput map(), filter(), reduce(), zip(), itd. Primeri nekih funkcija iz ove biblioteke mogu se naći

u dodacima A.1.

Iako Lua nije objektno-orjentisan jezik, on obezbeđuje sve objekte koji nam dopuštaju da implementiramo sistem objekata. Umesto klasa, u Lui pametna upotreba meta tabela može da kreira sistem klasa. Meta tabele mogu da kreiraju objekte zasnovane na prototipovima. Više o objektima u nastavku teksta.

5 Okruženja

Tokom godina, nastala su brojna okruženja za jezik Lua koja olakšavaju razvoj veb aplikacija. Korišćena su za razvoj mnogih poznatih sajtova i aplikacija, između ostalog i za Vikipedija templejting sistem, kao i kineski sajt za onlajn kupovinu Taobao.

Među najpopuilarnija Lua okruženja spadaju Lapis, Sailor, Luvit i Fengari.

Lapis je okruženje za pisanje veb aplikacija korišćenjem Lue ili Munskripta [3] i izvršava se u okviru distribucije endžinIksa (eng. Nginx) zvane OpenResti (eng. OpenResty). OpenResti izvršava Lua (odnosno MunSkript) koristeći LuaJIT. Lapis je po svojoj funkcionalnosti analogan Reils (eng. Rails) okruženju za Rubi (eng. Ruby) i Laravel okruženju za PHP. Omogućava HTML templating, jednostavno uvođenje middlewarea, upravljanje ORM modelima (uključujući i migracije istih) i slično.

Sailor okruženje je slično Lapisu ali razlikuje se po tome što uvodi dodatne, naprednije funkcionalnosti, od kojih je najvažnija mogućnost pisanja klijentskog koda u Lui, pomoću Lua virtuelne mašine implementirane u JavaScriptu. Još jedna prednost je što nije vezan za OpenResti, već ga je moguće izvršavati i na Apači, Endžiniksu, Mongus, Lajti, Savant i Luan (eng. Apache2, Nginx, Mongoose, Lighttpd, Xavante, Lwan redom) veb serverima. Naziv je inspirisan popularnom animiranom serijom zvanom ,,Sailor Moon", naime kada je autorka okruženja, ucenica jednog od autora jezika, počela da uči Luu, rešila je da ako ikada napiše projekat u njoj, zvaće se Sailor jer Lua znači mesec na portugalskom kao što je gore pomenuto. Značajna je i po tome što je učenica jednog od autora Lue[2].

Luvit je okruženje za Luu koje implementira API identičan Node.js okruženju. Asinhrone I/O operacije se oslanjaju na istu biblioteku kao u Node.js-u (libuv). Zbog ove dve stvari, kod pisan za Luvit izgleda veoma slično kodu Node.js-a, stoga može biti pogodna polazna tačka za JavaScript/Node programere koji žele da nauče Luu.

Fengari je implementacija Lua virtuelne mašine pisana u JavaScriptu. Izvršava se u veb pregledaču i omogućava izvršavanje Lua programa u tom okruženju. Lua kodu je dostupan kompletan API pregledača, uključujući i funkcije za manipulaciju DOM-a, pa može u potpunosti da zameni JavaScript pri pisanju klijentskog koda[3].

6 Instalacija

Lua je besplatna i pod MIT licencom. Na Linuks (eng. Linux) i Mek (eng. Mac) operativnom sistemu, Lua bi trebalo da je već instalirana ili postoje paketi za nju.

[3] eng. Moonscript — programski jezik koji se kompajlira u Lui, a sa čitljivijom i kraćom sintaksom od njene

```
# nacin 1:
sudo apt install lua5.3                          #Debian/Ubuntu systems
# yum install epel-release && yum install lua #RHEL/CentOS systems
# dnf install lua                                #Fedora 22+

# nacin 2:
# instalacija potrebnih paketa ako ih vec nema:
sudo apt install build-essential libreadline-dev # Debian/Ubuntu
# yum groupinstall "Development Tools" readline  # RHEL/CentOS systems
# dnf groupinstall "Development Tools" readline  # Fedora 22+

curl -R -O http://www.lua.org/ftp/lua-5.3.5.tar.gz #skida poslednju verziju
tar zxf lua-5.3.5.tar.gz # radi raspakivanje
cd lua-5.3.5             # pozicionira se u raspakovani folder
make linux test          # kompajlira i testira
sudo make install        # instalira
```

Kod 1: Instalacija iz terminala na linuxu

U slučaju instalacije, treba pratiti uputstva iz koda 6, s tim što se u zavisnosti od platforme 'linux' treba zameniti doslovno nekom od reči iz niza:

`aix, bsd, c89, freebsd, generic, macosx, mingw, posix, solaris`

U slučaju da nijedna reč nije odgovarajuća treba izabrati najsrodniju. Interpreter se može koristiti i bez poslednjeg koraka [4].

Za Windows se preporucuje praćenje instrukcija sa zvanične stranice[4]. U svrhu izbegavanja instalacije Lue moguće je koristiti onlajn interpreter[5].

7 Programiranje u Lui

Lua teži tome da bude fleksibilna, ali, takođe teži tome da bude mali jezik. Za ugradne jezike ovo je bitna osobina pošto se često koriste u uređajima koji imaju ograničene hardverske resurse. Kako bi se postigla ova dva suprotstavljena cilja, dodavanju novih karakteristika u jezik pristupa se ekonomično. Zbog toga Lua koristi malo mehanizama. A pošto ih je malo, oni moraju biti efikasni. Neki od takvih mehanizama su, na primer, tabele (opisane u poglavlju 7.1), funkcije prvog reda (opisane u poglavlju 7.2), zatvorenja (opisana u poglavlju 7.3), iteratori (opisani u poglavlju 7.4) i refleksivne mogućnosti. Da bi jezik bio što manji, umesto hijerarhije numeričkih tipova (realni, racionalni, celi), Lua ima samo brojeve u pokretnom zarezu dvostruke preciznosti kao tip vrednosti [8]. Lua ima ukupno 8 tipova: *nil*, *bool*, *brojevi*, *string*, *korisnički podaci*, *funkcije*, *niti* i *table*. U slučaju greške funkcija vraća nil i ono karakteriše odsustvo vrednosti. Nil nalazi primenu u brisanju promenljivih i oslobađanju memorije. Kada se promenljivoj dodeli nil, njena ranija vrednost biva izbrisana ukoliko ništa ne pokazuje na nju. Svaki tip može biti ,,kastovan" u bool, a način je dat u tabeli 1. U potpoglavljima biće više reči o nekim važnim konceptima ovog programskog jezika.

[4]zvanična strana: http://lua-users.org/wiki/BuildingLuaInWindowsForNewbies
[5]link online interpretera: http://lua-users.org/wiki/BuildingLuaInWindowsForNewbies

Tip	Vrednost	Bool vrednost
string	”čokolada”	true
string	””	true
number	28	true
number	0	true
nil		false

Tabela 1: Kastovanje u tip bool

7.1 Tabele

Tabele su dinamički kreirani asocijativni nizovi [6]. Tabele se sastoje iz parova ključ-vrednost. Ako su ključevi tabele numerički, tabela predstavlja niz. Ako su ključevi ne-numeričke ili mešovite vrednosti, tabela je rečnik. Kao ključ u tabeli se može koristiti bilo šta, osim nil. Bilo šta, uključujuci i nil, može biti vrednost. Program jedino preko referenci (ili pokazivača) njima manipuliše. Nema skrivenih kopija ili kreiranja novih tabela iza scene.

Tabela se kreira uz pomoć konstruktora. Nakon što je kreirana, tabelu treba dodeliti promenljivoj, u suprotnom na nju nije moguće referisati. Kada u programu ne postoji ni jedna referenca na neku tabelu, upravljač memorije će obrisati tu tabelu (i osloboditi memoriju koju je tabela zauzimala), tako da memorija koju je ta tabela zauzimala kasnije može biti ponovo upotrebljena. Tabela je relaciona struktura podataka koja skladišti vrednosti. Da bi se promenljiva sačuvala u tabeli, koristi se sledeća sintaksa:

```
table[key] = value
```

Primer 2 demonstrira kako napraviti tabelu, sačuvati vrednost sa ključem x, i kako tu vrednost izvući iz tabele:

```
k = "x"
a[k]=10          --> nov ulaz, sa kljucem key="x"
                                 i vrednoscu value=10
a[20] = "great"--> nov ulaz, sa kljucem key=20
                                 i vrednoscu value="great"
print(a["x"])   --> ispisuje 10
k = 20
print(a[k])     --> ispisuje "great"
a["x"] = a["x"] + 1    --> uvecava ulaz "x"
print(a["x"])   --> ispisuje 11
```

Kod 2: Primer čuvanja vrednosti u tabeli

Meta tabele i meta metodi

Meta tabela je standardna tabela u Lui koja sadrži skup meta metoda koji mogu da promene ponašanje tabela. Meta metode su funkcije sa specifičnim imenom koje se pozivaju kada Lua izvršava određene operacije kao sto su sabiranje, konkatenacija stringova, poređenje itd. Na primer, koristeći meta tabele i meta metode, možemo definisati kako Lua računa izraz $a + b$, gde su a i b tabele. Kad god Lua proba da sabere dve tabele, prvo proverava da li svaka od njih ima meta tabelu i da li ta meta tabela

[6] Asocijativni niz je niz koji može biti indeksiran ne samo brojevima, već i stringovima ili bilo kojim drugim vrednostima, osim nil.

ima *__add* polje. Ako Lua pronadje to polje, poziva odgovarajuću funkciju za računanje sume.

Bilo koja tabela može biti meta-tabela bilo koje druge tabele. Grupa povezanih tabela može da deli zajedničku meta-tabelu (koja opisuje njihovo zajedničko ponašanje), a takođe tabela može biti svoja sopstvena meta-tabela (tako da opisuje svoje individualno ponašanje) [7].

Kreiranje meta tabele

Za kreiranje meta-tabele neophodno je prvo kreirati običnu tabelu, a zatim njoj pridružiti odgovarajuće funkcije. U primeru 3 obična tabela je nazvana *meta*. Ovoj tabeli je pridružena funkcija koja se zove *__add*[7]. *__add* funkcija prima dva argumenta: levi argument je tabela sa poljem koje se zove *value*, a desni argument je broj:

```
meta = { }                                  -- kreira tabelu
meta.__add = function(left, right)          -- dodaje meta-metod
return left.value + right                   -- left je tabela
end
```

Kod 3: Kreiranje meta-tabele i dodavanje meta-metoda [7]

Zatim je napravljena nova tabela koja je nazvana *container*. *Container* tabela sadrži promenljivu nazvanu *value*, koja ima vrednost *5*:

```
container = {
value = 5
}
```

Kod 4: Tabela *container* [7]

Ako se u ovom trenutku pokuša sa dodavanjem broja 4 tabeli *container*, Lua izbacuje sintaksnu grešku. Ovo se dešava zato što nije moguće tabeli dodati broj. Kod koji uzrokuje grešku prikazan je u primeru 5.

```
result = container + 4                      -- neispravno
print ("result: " .. result)
```

Kod 5: Neispravno sabiranje tabele i broja [7]

Da bi ovaj kod bio ispravan potrebno je tabeli *container* dodati meta-tabelu, koja ima *__add* meta-metod. Funkcija koja se koristi da tabeli dodeli meta-tabelu zove se *setmetatable*. Ispravan kod dat je u primeru 6.

```
setmetatable(container, meta)               -- postavljanje meta-tabele
result = container + 4                      -- sada je ispravno
print ("result: " .. result)
```

Kod 6: Ispravno sabiranje tabele i broja [7]

7.2 Funkcije

Funkcije u Lui su vrednosti prve klase (eng. *first-class values*) sa odgovarajućim ,,leksičkim opsegom". Za funkciju se kaže da je vrednost prve klase ako ona ima ista prava kao i vrednosti poput brojeva i stringova. Funkcije mogu da se čuvaju u promenljivama (globalnim i lokalnim) i u poljima tabela, da se prosleđuju drugim funkcijama kao argumenti i da budu vraćene kao povratne vrednosti funkcija. Da funkcija ima ,,leksički opseg" znači da može pristupati promenljivama funkcija kojima je okružena. Ova

[7] *__add* je rezervisano ime za funkciju.

osobina omogućava da u Lui možemo da primenimo tehnike programiranja iz sveta funkcionalnih jezika kao i da program bude kraći i jednostavniji.

Iako su funkcije vrednosti, postoji izraz kojim se funkcija kreira - deklaracija funkcije obično izgleda ovako:

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

Kod 7: Deklaracija funkcije [7]

Deklaracija funkcije započinje ključnom rečju *function*, nakon koje sledi ime funkcije, a zatim lista parametara funkcije, koja može biti prazna ako funkciji nisu potrebni parametri. Nakon liste parametara piše se telo funkcije. Telo funkcije se završava navođenjem ključne reči *end*. Međutim, prethodni primer deklaracije funkcije samo je lepši način da se zapiše:

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

Kod 8: Dodela vrednosti tipa *function* promenljivoj foo [7]

To jest, definicija funkcije je, u stvari, naredba koja promenljivoj dodeljuje vrednost tipa *function*. Izraz sa desne strane operatora dodele se može posmatrati kao konstruktor za funkcije, baš kao što se vitičastim zagradama predstavlja konstruktor za tabelu. Rezultat takvog konstruktora funkcije zovemo anonimna funkcija. Iako se uglavnom funkcijama dodeljuju imena, postoje i situacije kada funkcije treba da ostanu anonimne. Na primer, biblioteka za tabele pruža funkciju *table.sort*, koja prima tabelu i sortira njene elemente. Ova funkcija treba da omogući više varijacija sortiranja: rastuće ili opadajuće, numeričko ili alfabetsko, tabele sortirane po ključu, itd. Umesto kucanja posebnog koda da bi bile omogućene sve varijacije sortiranja, *sort* pruza jedan (jedini) opcioni parametar, koji predstavlja funkciju za poređenje (engl. *order function*), koja vrši poređenje dve vrednosti - prima dva elementa i vraća da li prvi argument treba da bude pre drugog. Sledeći primer prikazuje gde je zgodno upotrebiti anonimnu funkciju. Na primer, neka je data tabela sa slogovima:

```
network = {
  {name = "grauna",  IP = "210.26.30.34"},
  {name = "arraial", IP = "210.26.30.23"},
  {name = "lua",     IP = "210.26.23.12"},
  {name = "derain",  IP = "210.26.23.20"},
}
```

Kod 9: Primer tabele [7]

Da bi se vrednosti u tabeli sortirale po imenu polja u obrnutom alfabetskom poretku, kod treba da izgleda ovako:

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

Kod 10: Sortiranje vrednosti tabele [7]

Funkciju, koja prima drugu funkciju kao argument, kao što je *sort*, zovemo funkcijom višeg reda (eng. *higher-order function*). Funkcije višeg reda predstavljaju mehanizam u programiranju, koji za kreiranje svojih argumenata često koristi anonimne funkcije. Funkcije višeg reda su posledica mogućnosti Lue da upravlja funkcijama kao vrednostima prve klase.

7.3 Zatvorenja

U Lui, promenljive koje su lokalne za neku funkciju su takođe dostupne u funkcijama koje su definisane unutar te funkcije, tj. unutar ugnježdenih definicija. Na primer, kada postoji neka funkcija koja vraća anonimnu funkciju, anonimna funkcija može da vidi lokalne promenljive funkcije kojom je okruzena. Međutim, nakon sto je anonimna funkcija vraćena, ona može da nadživi funkciju koja ju je vratila kreirajuci zatvorenje. Ovaj mehanizam dopušta pristup stanju funkcije omotača i nadpromenljivama[8], iako se funkcija omotac više ne izvršava. Zatvorenje je funkcija, plus sve što joj je potrebno da pristupi njenim nadpromenljivama korektno. Ono što je vrednost u Lui je zatvorenje, a ne funkcija. Funkcija, sama za sebe, samo je prototip za zatvorenje.

7.4 Iteratori

Iterator, konstrukcija koja omogćava prolazak kroz kolekciju, predstavlja se pomoću funkcije. Pri svakom pozivu, ta funkcija vraća naredni element iz kolekcije. U kodu 11 dat je primer iteratora nad listom koji vraća vrednost elemenata liste.

```
function values (t)
  local i = 0
  return function () i = i + 1; return t[i] end
end

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

Kod 11: Primer iteratora nad listom

Između uzastopnih poziva, iterator mora da pamti stanje u kom se nalazi kako bi znao da nastavi dalje odatle. U tu svrhu se koriste zatvorenja. [6].

U kodu 11 funckija omotač je values(). Pri svakom pozivu ona pravi zatvorenje (koje predstavlja sam iterator). To zatvorenje čuva svoje stanje u spoljašnjim promenljivama (*t, i, n*) tako da, svaki put kada se pozove, vraća naredni element iz liste t. Kada više nema vrednosti u listi, vraća nil.

Postoje i iteratori bez stanja (eng. *stateless iterators*). To su iteratori koji ne čuvaju sami svoja stanja, tako da možemo isti iterator iskoristiti u više petlji, bez potrebe da pravimo nova zatvorenja. U svakoj itreaciji, for petlja poziva svoj iterator sa dva argumenta: invarijantnim stanjem i kontrolnom promenljivom. Iterator bez stanja generiše naredni element na osnovu te dve vrednosti. U kodu 12 prikazan je jedan iterator bez stanja - ipairs()[7].

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

Kod 12: Primer iteratora bez stanja

Rezultat izvrsavanja ovog programa je:

[8]eng. upvalue – vrednost promenljive funkcije omotaca

```
1 one
2 two
3 three
```

Funkcija ipairs() vraća tri vrednosti: *gen*, *param*, i *state* (koje se nazivaju iteratorski triplet). *Gen* je funkcija koja generiše narednu vrednost u svakoj iteraciji. Ona vraća novo stanje (*state*) i vrednost u tom stanju. *Param* je trajni parametar *gen* funkcije i koristi se za pravljenje instance *gen* funkcije, npr. tabele. *State* je privremeno stanje iteratora koje se menja nakon svake iteracije, npr. to je indeks niza[10]. U kodu 13 su prikazane ove tri vrednosti.

```
a = {"one", "two", "three"}
gen, param, state = ipairs(a)
# rezultat izvrsavanja: function: 0x41b9e0  table: 0x1e8efb0  0
print(gen(param, state))
# rezultat izvrsavanja: 1 one
```

Kod 13: Primer iteratora bez stanja (nastavak)

7.5 Primer jednostavnog koda

U ovom poglavlju će biti dat primer jednog jednostavnog koda u kom se može videti način na koje se koriste prethodno opisani elementi ovog programskog jezika. U primeru 14 je prikazano kako je moguće naći sve faktore broja koji se unosi sa standardnog ulaza. Od korisnika se prvo traži da unese broj, zatim se on učitava pomoću ugrađene funkcije *io.read()* i poziva se korisnički definisana funkcija koja vraća tabelu svih faktora. U okviru te funkcije prvo je potrebno deklarisati praznu tabelu u koju će biti dodavani faktori. Zatim se u *for* petlji proveravaju svi brojevi od 1 do korena prosleđenog broja, pri čemu je korak u iteriranju jednak 1. Ako taj broj deli prosleđeni broj, onda su on i broj koji se dobija deljenjem prosleđenog broja tim brojem faktori i dodaju se u tabelu pomoću funkcije *insert()*. Zatim se tabela sortira i vraća kao rezultat funkcije. Na kraju, pri povratku iz funkcije, prolazi se kroz sve elemente tabele pomoću iteratora *ipairs()* i oni se ispisuju na standardni izlaz. U ovom primeru se, takođe, može videti da se lokalne promenljive deklarišu ključom rečju *local*. A prikazano je i pisanje jednolinijskih i višelinijskih komentara.

```
function get_all_factors(number)
    --[[--
      Ova funkcija vraca sve faktore datog broja.
      Preciznije, vraca tabelu koja sadrzi sve faktore.
    --]]--

    local factors = {}  -- Tabela u kojoj cuvamo faktore
    for possible_factor=1, math.sqrt(number), 1 do
        local remainder = number % possible_factor
        if remainder == 0 then
           local factor1, factor2 = possible_factor, number/possible_factor
           table.insert(factors, factor1)
           table.insert(factors, factor2)
        end
    end
    table.sort(factors)
    return factors
end

-- ovako se ispisuje poruka na standardni izlaz:
print("Unesite broj cije faktore zelite da izracunate:")
-- ovako se ucitava broj sa standardnog ulaza:
num = io.read("*n")
-- ovako se poziva funkcija:
```

```
factors_of_num = get_all_factors(num)
-- pomocu ipairs() iteratora se prolazi kroz kolekciju i ispisuje
-- se svaki od faktora prosledjenog broja
for i, factor in ipairs(factors_of_num) do
   print(factor)
end
```

Kod 14: Primer izračunavanja faktora broja

8 Zaključak

U radu je pokazano kako Lua, iako je nastala za lokalne potrebe, danas ima primenu u svetu. Njena mala sintaksa navodi programere da raspolazući jednostavnim komponentama stvaraju efikasne algoritme, a jednostavnost sintakse se čuva iz verzije u verziju. Uprkos tome spektar načina njenog primenjivanja je širok. Lua se koristi u međusobno vrlo različitim projektima i na međusobno vrlo različite načine. Sve ove osobine svedoče tome da je Lua dobar izbor programskog jezika ne samo za spajanje raznorodnih projekata već i za započinjanje istih. Pored trenutnih mehanizama koje ima, u budućnosti bi ovaj jezik mogao dovesti do ideja za efikasniju implementaciju struktura podataka posredstvom tabela.

Literatura

[1] Adobe. Adobe Photoshop Lightroom , 2019. on-line at: https://www.adobe.io/apis/creativecloud/lightroomclassic.html.

[2] Etiene Dalcol. Sailor Project, 2014-2015. on-line at: http://sailorproject.org/main/about.

[3] Fengari. Fengari. on-line at: https://fengari.io/.

[4] Roberto Ierusalimschy. Lua, 2003-2004. on-line at: https://www.lua.org/start.html.

[5] Roberto Ierusalimschy. Lua - list of news, 2003-2004. on-line at: https://www.lua.org/news.html.

[6] Roberto Ierusalimschy. Lua.org, 2003-2004. on-line at: https://www.lua.org/pil/7.1.html.

[7] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, Rio de Janeiro, 2006.

[8] Roberto Ierusalimschy. Programming with multiple paradigms in Lua. *Proceedings of the 18th International Workshop on Functional and (Constraint) Logic Programming*, page 5–13, 2009. on-line at: www.inf.puc-rio.br/%7Eroberto/docs/ry09-03.pdf.

[9] Kepler Project. Kepler Project, 2003. on-line at: https://keplerproject.github.io/cgilua/.

[10] Roman Tsisyk. Lua Fun Documentation, 2013-2017. iterators: https://luafun.github.io/under_the_hood.html, map: https://luafun.github.io/transformations.html#fun.map, filter: https://luafun.github.io/filtering.html, zip: https://luafun.github.io/compositions.html#fun.zip.

A Dodatak

A.1 Korišćene funkcije

1. fun.map(*fun, gen, param, state*)

   map() prihvata funkciju *fun* i vraća novi iterator koji je nastao primenjivanjem funkcije *fun* na svaki od elemenata tripletskog iteratora *gen, param, state*. Mapiranje se vrši u prolazu kroz kolekciju, bez baferisanja vrednosti. Primer je dat u kodu 15. [10]

```
> each(print, map(function(x) return 2 * x end, range(4)))
2
4
6
8
```

Kod 15: Primer funkcije map()

2. fun.filter(*predicate, gen, param, state*)

   filter() prihvata predikat *predicate* na osnovu koga filtrira iterator predstavljen sa *gen, param, state*. Vraća novi iterator za elemete koji zadovoljavaju predikat. Primer je dat u kodu 16. [10]

```
> each(print, filter(function(x) return x % 3 == 0 end, range(10)))
3
6
9
```

Kod 16: Primer funkcije filter()

3. fun.zip(iterator1, iterator2, ...)

   zip() prihvata listu iteratora i vraća novi iterator čija i-ta vrednost sadrži i-ti element iz svakog od prosleđenih iteratora. Povratni iterator je iste dužine kao najkraći prosledjeni iterator. Primer je dat u kodu 17. [10]

```
> each(print, zip(range(5), {'a', 'b', 'c'}, rands()))
1       a       0.57514179487402
2       b       0.79693061238668
3       c       0.45174307459403
```

Kod 17: Primer funkcije zip()