

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, ivana.jordanov47@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 da se pokaže 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	2
2	Nastanak jezika	2
3	Primena	3
4	Podržane paradigme	4
5	Razvojna okruženja	4
6	Instalacija	5
7	Programiranje u Lui	6
8	Zaključak	12
	Literatura	12
A	Dodatak	13

1 Uvod

Programski jezik Lua spada u dinamički tipizirane, *gradivne* (eng. embeddable) ¹ skript jezike. Lua podržava proceduralno, objektno orijentisano i funkcionalno programiranje, kao i *programiranje vođeno podacima* (eng. data-driven programming) ²

Ovaj programski jezik je nastao kao potreba za fleksibilnim programskim jezikom, koji će omogućiti jednostavno korišćenje nesrodnih mehanizama različitih programskih paradigmi. Zbog toga, Lua danas ima široku primenu u različitim oblastima programiranja. Koji su to mehanizmi i kako je Lua nastala opisano je u poglavlju 2, dok se o njenoj primeni govori u poglavlju 3. Načini na koje su podržane funkcionalna i objektno-orijentisana paradigma opisani su u poglavlju 4. Za programiranje u Lui postoje brojna okruženja, o čemu se govori u poglavlju 5, a instrukcije za instalaciju su date u poglavlju 6. Jednostavnost programskog jezika je vidljiva kroz kodove koji demonstriraju ideju na kojoj je zasnovano programiranje u Lui (poglavlje 7).

2 Nastanak jezika

Programski jezik Lua je nastao 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.

2.1 Istorijski kontekst

Zbog tadašnjih brazilskih zakona, koji su striktno ograničavali uvoz softvera i hardvera, favorizujući domaće proizvode i otežavajući uvoz iz inostranstva, nastala je 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 s. Amerike.

2.2 Prethodnici jezika Lua

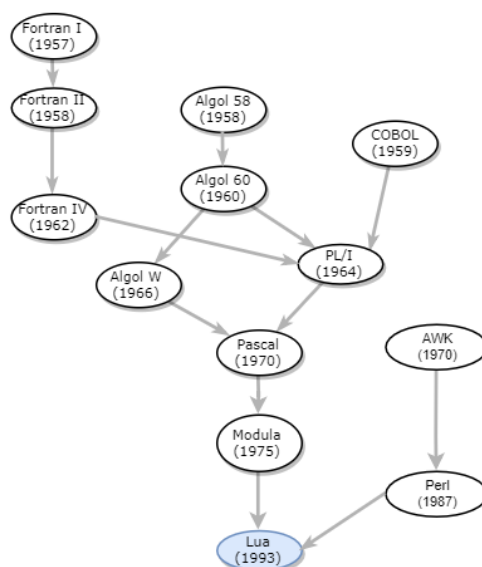
Prethodnici Lue, dva mala programska jezika zvana DEL (Data Entry Language) i SOL (Simple Object Language, čija skraćena u prevodu označava sunce, što je bila direktna inspiracija za naziv Lua), nastali su 1992. za potrebe brazilskog naftnog giganta Petrobrasa. Svrha naručenog grafičkog interfejsa je bila da olakša, ubrza i učini otpornijim na greške unos ogromne količine numeričkih podataka koje su morale biti unete u zastarelom formatu nasleđenom iz doba bušenih kartica. DEL je nastao kako bi se olakšalo pisanje ovih grafičkih interfejsa, kao deklarativni jezik koji je opisivao svojstva i ograničenja polja za unos podataka i entiteta kojima su ta polja pripadala. SOL je nastao kako bi se olakšalo konfigurisanje litoloških podataka.

¹*Gradivni jezici* predstavljaju klasu jezika koji se koriste da prošire ili utiču na ponašanje postojeće aplikacije.

²*Programiranje vođeno podacima* je programska paradigma u kojoj postoji jasna razlika između koda i struktura podataka na koje kod deluje.

Međutim, zbog potreba za dodatnim mogućnostima u okviru DEL-a i SOL-a, odlučeno je da se napravi nov programski jezik koji će da sadrži naredbe dodele, kontrole toka, podrutine itd. Portabilnost, jednostavna sintaksa i semantika i opis podataka u SQL-u bili su prioriteta koje je ovaj jezik trebalo da zadovolji. Mnogi koncepti u Lui pozajmljeni su iz drugih programskih jezika, uključujući Modulu, CLU, C++, SNOBOL i Awk. Na slici 1 je prikazan uticaj nekih takođe bitnih jezika za nastanak Lue.

U julu 1993, Valdermar je dovršio prvu implementaciju Lue, koja je prevedena korišćenjem lex i yacc alata. 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.



Slika 1: Razvojno stablo

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 CGI Lua, alat za pravljenje dinamičkih veb stranica i manipulisanje podacima prikupljenih veb formama. U suštini, CGI Lua 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 efekti, za dodavanje stavki u menije i dijaloge, manipulaciju metapodacima i kreiranje novih tipova veb galerija [1].

Danas se Lua 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 specifičnih 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 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()`, `zip()` itd. Primeri funkcija ove biblioteke nalaze se u dodatku A.1.

Iako Lua nije objektno-orijentisan jezik, on obezbeđuje sve objekte koji omogućavaju implementaciju 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 i tabelama sledi u nastavku teksta.

5 Razvojna 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 sistem šablona, kao i kineski sajt za onlajn kupovinu Taobao. Među najpopularnija Lua okruženja spadaju Lapis, Sailor, Luvit i Fengari, i oni će biti opisani u nastavku.

Lapis je okruženje za pisanje veb aplikacija korišćenjem Lue ili *Mun skripta* (eng. Moonscript)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 sistem šablona, jednostavno uvođenje *posrednog softvera* (eng. middleware, MW)⁴, 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. Apache, Nginx, Mongoose, Lighttpd, Xavante, Lwan) veb serverima. Naziv je inspirisan popularnom animiranom serijom zvanom „Sailor Moon”. Naime, kada je autorka okruženja, učenica 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[4] okruženju. Asinhrono 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 besplatan softver *otvorenog koda* (eng. open source), distribuiran pod uslovima MIT licence. Iako Lua spada u programske jezike koji se interpretiraju, Lua uvek prvo kompajlira izvorni kod pre nego što ga pokrene. Lua se pokreće na Linux, Windows i Mac operativnim sistemima, a takođe i na Android, iOS, BREW, Symbian operativnim sistemima, kao i na mnogim drugim. U nastavku je opisano kako se Lua instalira na Linux, Mac i Windows operativnim sistemima, tim redom.

6.1 Instalacija na Linux i Mac operativnim sistemima

Lua paket je dostupan u zvaničnim repozitorijumima Linux distribucije. Poslednja verzija Lue se instalira korišćenjem odgovarajućeg paket

³ *Mun skript* je programski jezik koji se kompajlira u Lui, sa čitljivijom i kraćom sintaksom od njene.

⁴ Posredni softver je višenamenski softver koji pruža usluge aplikacijama izvan onoga što nudi operativni sistem. Bilo koji softver između jezgra i korisničke aplikacije može biti posredni softver.

menadžera. Za instalaciju na Linux-u treba pratiti uputstva iz koda 1, a u zavisnosti od platforme, reč **linux** treba zameniti nekom od sledećih reči: **aix**, **bsd**, **c89**, **freebsd**, **generic**, **macosx**, **mingw**, **posix**, **solaris**. U slučaju da ni jedna reč nije odgovarajuća, treba izabrati najsirodniju.

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

-- nacin 2:
-- instalacija odgovarajucih paketa ako je to potrebno:
sudo apt install build-essential libreadline-dev -- Debian/Ubuntu
-- yum groupinstall "Development Tools" readline -- RHEL/CentOS
-- dnf groupinstall "Development Tools" readline -- Fedora 22+

-- instalacija poslednje verzije Lue:
curl -R -O http://www.lua.org/ftp/lua-5.3.5.tar.gz
tar xzf 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 korišćenjem terminala na Linux OS

Instalacija na Mac-u se razlikuje samo u poslednjoj liniji koda 1, gde umesto nje treba navesti **make macosx test**. Osim **curl** koristi se i **wget**.

6.2 Instalacija na Windows operativnom sistemu

Za instalaciju Lue na Windows operativnom sistemu potreban je TDM-GCC kompajler, koji se instairu u nekoliko koraka opisanih u nastavku. Prvo treba preuzeti paket za instalaciju `tdm-gcc-4.9.2.exe`, gde oznaka 4.9.2 predstavlja verziju kompajlera, pa je to moguće promeniti tako da bude odgovarajuća. Zatim, treba napraviti radni direktorijum i u njega smestiti prethodno skinuti paket. Nakon toga, treba preuzeti izvorni kod Lue `lua-5.3.0.tar.gz`, smestiti ga u radni direktorijum i raspakovati ga. Dalje je potrebno napraviti *šel skriptu* (*eng.* shell script) `build.cmd` u kojoj treba napisati na koji način će biti instalirana binarna verzija. Tekst koji treba da bude u tom fajlu se može videti u dodatku A.2. Zatim treba pokrenuti taj skript i sačekati da se instalacija završi. Nakon toga, biće kreiran folder `lua`, u čijem bin folderu se nalazi fajl `lua.exe`, interpreter za Luu, a time je instalacija završena.

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 tabele (poglavlje 7.1), funkcije prvog reda (poglavlje 7.2), zatvorenja (poglavlje 7.3), iteratori (poglavlje 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*, što karakteriše odsustvo vrednosti. *Nil* nalazi primenu u brisanju promenljivih i oslobađanju memorije, s obzirom da Lua podržava automatsko upravljanje memorijom⁵. Kada se promenljivoj dodeli *nil*, njena ranija vrednost biva izbrisana ako ništa ne pokazuje na nju. Svaki tip može biti „kastovan” u *bool*, a načini na koje se neki od tipova kastuju dat je u tabeli 1. U nastavku će biti reči o pomenutim mehanizmima (konceptima) ovog jezika, a ukoliko čitaoca zanima kako ti koncepti mogu da se kombinuju prilikom pisanja programa, može pogledati primer koda u dodatku A.3.

Tabela 1: Kastovanje u tip *bool*

Tip	Vrednost	Bool vrednost
<i>string</i>	”čokolada”	<i>true</i>
<i>string</i>	””	<i>true</i>
<i>number</i>	28	<i>true</i>
<i>number</i>	0	<i>true</i>
<i>nil</i>		<i>false</i>

7.1 Tabele

Tabele su dinamički kreirani asocijativni nizovi⁶. 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, uključujući i *nil*, može biti vrednost. Program manipuliše tabelama isključivo preko referenci (ili pokazivača). 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 zauzimala), tako da memorija koju je ta ona zauzimala kasnije može biti ponovo upotrebljena. Tabela je relaciona struktura podataka koja skladišti vrednosti. Da bi se vrednost sačuvala u tabeli, koristi se sintaksa data u kodu 2.

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

Kod 2: Primer čuvanja vrednosti u tabeli

Kod 3 demonstrira kako napraviti tabelu i sačuvati vrednost sa ključem *x*. Na kraju je u istom kodu prikazano 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
```

⁵Automatsko upravljanje memorijom koristi *sakupljač otpadaka* (*eng.* garbage collection) koji se s vremena na vreme aktivira i krene da sakuplja objekte (tabele, podatke korisnika, funkcije, niti, stringove itd.) koji više nisu pristupačni u programu.

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

```

k = 20
print(a[k])           -- ispisuje "great"
a["x"] = a["x"] + 1  -- uvecava ulaz "x"
print(a["x"])        -- ispisuje 11

```

Kod 3: 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 što su sabiranje, konkatencija stringova, poređenje itd. Na primer, pomoću meta-tabela i meta-metoda se može definisati kako da 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 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 kodu 4 obična tabela je nazvana *meta*. Ovoj tabeli je pridružena funkcija koja se zove `__add`⁷. Funkcija `__add` prima dva argumenta: levi argument je tabela sa poljem koje se zove *value*, a desni argument je broj.

```

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

```

Kod 4: Kreiranje meta-tabele i dodavanje meta-metoda [7]

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

```

container = {
  value = 5
}

```

Kod 5: 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 kodu 6.

```

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

```

Kod 6: 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 kodu 7.

⁷`__add` je rezervisano ime za funkciju.


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

Kod 7: 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 tabele, da se prosleđuju drugim funkcijama kao argumenti i da budu vraćene kao povratne vrednosti funkcija. Da funkcija ima „leksički opseg” znači da može pristupiti promenljivama funkcija kojima je okružena. Ova osobina omogućava da u Lui mogu da se primene 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 bilo koje funkcije ima oblik prikazan u kodu 8, u kome je dat primer deklaracije funkcije *foo*.

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

Kod 8: Deklaracija funkcije [7]

Deklaracija funkcije započinje ključnom rečju *function*, nakon koje sledi ime funkcije, a zatim lista parametara funkcije. Lista parametara funkcije može biti i prazna ako funkciji nisu potrebni parametri. Nakon liste parametara piše se telo funkcije. Telo funkcije se završava navođenjem ključne reči *end*. Međutim, primer deklaracije dat u kodu 8, samo je lepši način da se zapiše kod koji je dat u kodu 9.

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

Kod 9: Dodela vrednosti tipa *function* promenljivoj *foo* [7]

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 zove se 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* pruža jedan (jedini) opcioni parametar, koji predstavlja *funkciju za poređenje* (eng. order function), koja vrši poređenje dve vrednosti - prima dva elementa i vraća da li prvi argument treba da bude pre drugog. U kodovima 10 i 11 ilustrovana je upotreba anonimne funkcije.

```
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 10: Primer tabele [7]

```

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

```

Kod 11: Sortiranje vrednosti tabele [7]

Funkcija, koja prima drugu funkciju kao argument, kao što je *sort*, zove se *funkcija 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 Lua 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 ugnježenih definicija. Primer koji ovo oslikava jeste kada postoji neka funkcija koja vraća anonimnu funkciju. Anonimna funkcija može da „vidi” lokalne promenljive funkcije kojom je okružena. Međutim, pošto je anonimna funkcija vraćena, ona može da nadživi postojeću funkciju. Kada se vrati anonimna funkcija kao povratna vrednost neke funkcije, ona kreira zatvorenje. Ovaj mehanizam dopušta pristup stanju spoljašnje funkcije (koje ovo zatvorenje obuhvata), iako se ta funkcija više ne izvršava. Ovo je prikazano u kodu 12.

```

function newCounter ()
    local i = 0 -- lokalna promenljiva za newCounter() funkciju
    return function () -- anonimna funkcija

-- anonimna funkcija je kreirana unutar funkcije newCounter (),
-- pa ona vidi sve njene lokalne promenljive

-- kada napravi zatvorenje, ova funkcija pamti stanje
-- funkcije newCounter ()

        i = i + 1
        return i
    end

end

c1 = newCounter()
print(c1()) --> 1
print(c1()) --> 2

```

Kod 12: Primer zatvorenja

Ovde anonimna funkcija koristi eksternu lokalnu promenljivu *i* da održi svoj brojač. Međutim, kada se pozove anonimna funkcija, *i* je izvan *do-meta* (*eng.* scope), zato što je funkcija koja je kreirala tu promenljivu (funkcija *newCounter*) završila sa radom. Ipak, Lua razrešava tu situaciju korektno, koristeći koncept zatvorenja. Zatvorenje je funkcija, plus sve što joj je potrebno da bi pristupila njenim eksternim promenljivama. Ako se funkcija *newCounter* pozove ponovo, ona će napraviti novu lokalnu promenljivu *i*, pa će nastati novo zatvorenje koje onda deluje nad tom novom promenljivom, kao što je prikazano u kodu 13.

```

c2 = newCounter()
print(c2()) --> 1
print(c1()) --> 3
print(c2()) --> 2

```

Kod 13: Primer zatvorenja (nastavak)

Dakle, `c1` i `c2` su različita zatvorenja nad istom funkcijom, gde svako deluje na nezavisnu instancu lokalne promenljive `i`. Ono što je vrednost u `Lui` je zatvorenje, a ne funkcija. Funkcija, sama za sebe, je samo prototip za zatvorenje.

7.4 Iteratori

Iterator, konstrukcija koja omogućava prolazak kroz kolekciju, predstavlja se pomoću funkcije. Pri svakom pozivu, ta funkcija vraća naredni element iz kolekcije. U kodu 14 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 14: 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 14 funkcija 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 se isti iterator može iskoristiti u više petlji, bez potrebe za pravljenjem novih zatvorenja. U svakoj iteraciji, 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 15 prikazan je jedan iterator bez stanja - `ipairs()`[7].

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

Kod 15: Primer iteratora bez stanja

Rezultat izvršavanja ovog programa je:

```
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 16 su prikazane ove tri vrednosti.

```

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

```

Kod 16: Primer iteratora bez stanja (nastavak)

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, raspolažući jednostavnim komponentama, stvaraju efikasne algoritme. 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 o 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] Etienne Dalcol. Sailor Project, 2014-2015. on-line at: <http://sailorproject.org/main/about>.
- [3] Fengari. Fengari. on-line at: <https://fengari.io/>.
- [4] Node.js foundation. Node.js , 2009. on-line at: <https://nodejs.org/en/about/>.
- [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 Neke funkcije iz biblioteke Lua Fun

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

`map()` prihvata funkciju *fun* i vraća novi iterator koji je nastao primjenjivanjem 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 17. [10]

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

Kod 17: 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 18. [10]

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

Kod 18: 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 prosleđeni iterator. Primer je dat u kodu 19. [10]

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

Kod 19: Primer funkcije `zip()`

A.2 Instalacija na Windows operativnom sistemu - sadržaj `build.cmd` fajla

Prilikom instalacije Lue na Windows operativnom sistemu potrebno je napisati šel skriptu `build.cmd` kojom se opisuje na koji način će biti instalirana binarna verzija. U kodu 20 je prikazano šta je potrebno napisati u tom fajlu.

```
@echo off
:: =====
:: file build.cmd
:: =====
setlocal
:: you may change the following variable's value
:: to suit the downloaded version
set lua_version=5.3.0

set work_dir=%~dp0
:: Removes trailing backslash
```

```

:: to enhance readability in the following steps
set work_dir=%work_dir:~0,-1%
set lua_install_dir=%work_dir%\lua
set compiler_bin_dir=%work_dir%\tdm-gcc\bin
set lua_build_dir=%work_dir%\lua-%lua_version%
set path=%compiler_bin_dir%;%path%

cd /D %lua_build_dir%
mingw32-make PLAT=mingw

echo.
echo **** COMPILATION TERMINATED ****
echo.
echo **** BUILDING BINARY DISTRIBUTION ****
echo.

:: create a clean "binary" installation
mkdir %lua_install_dir%
mkdir %lua_install_dir%\doc
mkdir %lua_install_dir%\bin
mkdir %lua_install_dir%\include

copy %lua_build_dir%\doc\*. * %lua_install_dir%\doc\*. *
copy %lua_build_dir%\src\*. exe %lua_install_dir%\bin\*. *
copy %lua_build_dir%\src\*. dll %lua_install_dir%\bin\*. *
copy %lua_build_dir%\src\luaconf.h %lua_install_dir%\include\*. *
copy %lua_build_dir%\src\lua.h %lua_install_dir%\include\*. *
copy %lua_build_dir%\src\lualib.h %lua_install_dir%\include\*. *
copy %lua_build_dir%\src\lauxlib.h %lua_install_dir%\include\*. *
copy %lua_build_dir%\src\lua.hpp %lua_install_dir%\include\*. *

echo.
echo **** BINARY DISTRIBUTION BUILT ****
echo.

%lua_install_dir%\bin\lua.exe -e"print [[Hello!]];
print[[Simple Lua test successful!!!]]"

echo.

pause

```

Kod 20: Sadržaj build.cmd fajla

A.3 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 kodu 21 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 deklarirati 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 deklariraju ključom rečju *local*. A prikazano je i pisanje jednolinijskih i višelinijjskih komentara.

```

function get_all_factors(number)
  --[[--
  Ova funkcija vraća sve faktore datog broja.

```

```

    Preciznije, vraća tabelu koja sadrži sve faktore.
--]]--

local factors = {} -- Tabela u kojoj čuvamo 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 čije faktore želite da izračunate:")
-- ovako se učitava broj sa standardnog ulaza:
num = io.read("*n")
-- ovako se poziva funkcija:
factors_of_num = get_all_factors(num)
-- pomoću 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
end

```

Kod 21: Primer izračunavanja faktora broja