

Programski jezik Lua

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

Jovana Pejkić, Jana Jovičić, Katarina Rudinac, Ivana Jordanov
kontakt 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 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, a kako se danas koristi i koje karakteristike poseduje. U te svrhe rad obuhvata opis njegovih fundamentalnih mehanizama, samim tim i nacina kojim se složeni koncepti različitih programskih paradigmi posredstvom jednostavne semantike i svega 8 tipova podataka postižu u njemu.

Sadržaj

1	Uvod	2
2	Nastanak	2
3	Primena	3
4	Podržane paradigme	4
5	Okruženja	4
6	Instalacija	5
7	Programiranje u Lui	5
7.1	Tabele	6
7.2	Funkcije	8
7.3	Zatvorenja	10
7.4	Iteratori	10
8	Zaključak	11
	Literatura	11
A	Dodatak	12
A.1	Korišćene funkcije	12

1 Uvod

Lua spada u dinamički tipizirane, gradivne (eng. embeddable¹) skript jezike. Podržava proceduralno, objektno orjentisano, funkcionalno i programiranje vođeno podacima². 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 funkcionalna i objektno orjentisana paradigma biće spomenuti u poglavlju 4. Kao rezultat širenja brojna su okruženja za Lua 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

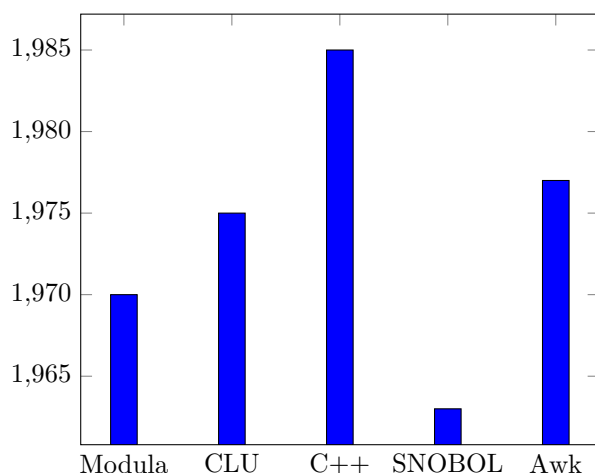
Programski jezik Lua nastao je 1993. na Katoličkom univerzitetu u Rio de Žaneiru u Brazilu, i u prevodu sa portugalskog znači mesec. U okviru Tecgraf-a, grupe za razvoj tehnologija u računarskoj grafici (eng. Computer Graphics Technology Group), pri pomenutom univerzitetu, trojica naučnika zvanih Roberto Jeruzalinski, 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, klikanjem 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

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

²programiranje vođeno 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 konfigurirati način ispisa. SOL je nastao kako bi se konfiguriranje 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 clici 2).

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 [5].

Ako se Lua upotrebljava u aplikaciji kao skript jezik, za njeno konfigurisanje u skladu sa domenom date aplikacije 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 dinamičkim veb stranicama i manipulisanje podacima prikupljenih putem veb formi. CGI Lua predstavlja apstrakciju za Veb server [7].

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, itd.

Treća mogućnost za upotrebu Lua jeste 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 Lua rade nad standardnom proceduralnom semantikom, što omogućuje laku integraciju između njih. Prema tome, većina programa napisanih u Lui su proceduralni, s tim što uključuju i korisne tehnike iz drugih paradigmi [6].

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 nam obezbeđuje sve objekte koji nam dopuštaju da implementiramo sistem objekata. Umesto klase, u Lui pametna upotreba meta tabela može da kreira sistem klase. Meta tabele mogu da kreiraju objekte bazirane 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 najpopularnija Lua okruženja spadaju Lapis, Sailor, Luvit i Fengari.

Lapis je okruženje za pisanje veb aplikacija korišćenjem Lua ili Munskripta³ 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 Re-

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

ils (eng. Rails) okruženju za Rubi (eng. Ruby) i Laravel okruženju za PHP. Omogućava HTML templating, jednostavno uvođenje middleware-a, 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, Endžiniks, 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, 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[1].

Luvit je okruženje za Luu koje implementira API identičan Node.js 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[2].

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.

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

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

U slučaju da nijedna nije odgovarajuća treba izabrati najsrodniju. Interpreter se može koristiti i bez poslednjeg koraka.[3]

Za Windows se preporučuje praćenje instrukcija sa zvanične stranice⁴. U svrhu izbegavanja instalacije Lue moguće je koristiti onlajn interpreter⁵.

7 Programiranje u Lui

Lua teži tome da bude fleksibilna, ali, takođe teži tome da bude mali jezik - i u pogledu implementacije i u terminima specifikacije. 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 se pristupa ekonomično. Zbog toga Lua koristi malo mehanizama. A pošto ih je malo, oni moraju biti efikasni. Neki od takvih mehanizama su, na primer, tabele, funkcije prvog reda, korutine i refleksivne mogućnosti. Da bi jezik bio što manji, umesto

⁴zvanična strana: <http://lua-users.org/wiki/BuildingLuaInWindowsForNewbies>

⁵link online interpretera: <http://lua-users.org/wiki/BuildingLuaInWindowsForNewbies>

```

# 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 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 iz terminala na linuxu

hijerarhije numeričkih tipova (realni, racionalni, celi), Lua ima samo brojeve u pokretnom zarezu dvostruke preciznosti kao tip vrednosti [6]. Lua ima ukupno 8 tipova: *nil*, *bool*, *brojevi*, *string*, *korisnicki 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 reči o tabelama i funkcijama kao i o nekim važnim konceptima vezanim za njih.

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

Jedina struktura podataka koju Lua pruža je tabela. Tabele su dovoljno fleksibilne da se pomoću njih mogu implementirati druge strukture podataka, kao što su liste, redovi ili stekovi.

Lua nije objektno-orijentisani jezik, te nema podršku za objekte. Međutim, sistem objekata se može implementirati korišćenjem tabela i meta-tabela.

Tabele su dinamički kreirani asocijativni nizovi⁶. One su u osnovi rečnik ili niz. 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. Tabele se skladište po referenci, a ne po vrednosti, i program jedino preko referenci

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

(ili pokazivača) njima manipuliše. Nema skrivenih kopija ili kreiranja novih tabela iza scene.

Tabela se kreira uz pomoć konstruktora, koji se označava pomoću otvorene i zatvorene vitičaste zagrade (primer 3). Nakon što je kreirana, tabelu treba dodeliti promenljivoj, u suprotnom na nju nije moguće referisati. Sledeći kod kreira novu tabelu i dodeljuje je promenljivoj `a`:

```
a = {} -- kreira tabelu i stavlja njenu referencu u promenljivu a
```

Kod 2: Funkcije

Skladištenje vrednosti

Tabela je relaciona struktura podataka koja skladišti vrednosti. Da bi se promenljiva sačuvala u tabeli, koristi se sledeća sintaksa:

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

Kod 3: Čuvanje vrednosti *value* u tabeli [5]

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 ona može biti kasnije ponovo upotrebljena.

Ključ tabelle može biti bilo kog tipa (pa čak i druga tabela), osim *nil*, i svaka tabela može da čuva vrednosti različitih tipova indeksa. Naredni primer demonstrira kako napraviti tabelu, sačuvati vrednost sa ključem `x`, i kako tu vrednost izvući iz tabelle:

```
a = {} -- prazna tabela
-- kreira se 1000 novih ulaza
for i=1,1000 do a[i] = i*2 end
print(a[9]) --> 18
a["x"] = 10
print(a["x"]) --> 10
print(a["y"]) --> nil
```

Kod 4: Kreiranje tabelle, dodeljivanje i vraćanje vrednosti [5]

U poslednjoj liniji primera 4 je prikazano da polja tabelle dobijaju vrednost *nil* ako nisu inicijalizovana. Takođe, ako se ne dodeli vrednost ključu u tabeli, podrazumevana vrednost je *nil*, a polju tabelle se može dodeliti vrednost *nil* ako se želi da ono bude obrisano.

Ako je ključ za tabelu tipa string, za pristup tabeli se može koristiti tačka sintaksa (eng. *dot syntax*). Lua podržava ovu reprezentaciju pružajući *a.ime* kao lepši zapis za [*"ime"*]. Dakle, poslednje tri linije prethodnog koda (primera 4) mogu se napisati i ovako:

```
a.x = 10 -- isto kao i a["x"] = 10
print(a.x) -- isto kao i print(a["x"])
print(a.y) -- isto kao i print(a["y"])
```

Kod 5: Primer čuvanja vrednosti u tabeli [5]

Meta tabelle i meta metodi

Meta tabela je standardna tabela u Lua-i koja sadrži skup meta metoda koji mogu da promene ponašanje tabelle. 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,

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 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) [5].

Kreiranje meta tabele

Za kreiranje meta-tabele neophodno je prvo kreirati običnu tabelu, a zatim njoj pridružiti odgovarajuće funkcije. U primeru 6 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 6: Kreiranje meta-tabele i dodavanje meta-metoda [5]

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

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

Kod 7: Tabela *container* [5]

Ako se u ovom trenutku pokuša sa dodavanjem broja 4 tabeli *container*, Lua izbacuje sintakсну grešku. Ovo se dešava zato što nije moguće tabeli dodati broj. Kod koji uzrokuje grešku prikazan je u primeru 8.

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

Kod 8: Neispravno sabiranje tabele i broja [5]

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 9.

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

Kod 9: Ispravno sabiranje tabele i broja [5]

7.2 Funkcije

Funkcije u Lua-i 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)

⁷ `__add` je rezervisano ime za funkciju.

i u poljima tabela, da se prosleđuju drugim funkcijama kao argumenti i da budu vraćene kao povratne vrednosti funkcija. Da funkcija ima "leksički opsegžnači da može pristupati promenljivama funkcija kojima je okružena. Ova osobina omogućava da u Lua-i 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 10: Deklaracija funkcije [5]

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 11: Dodela vrednosti tipa *function* promenljivoj foo [5]

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* pruža 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 12: Primer tabele [5]

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 13: Sortiranje vrednosti tabele [5]

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 Lua-e 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. Na primer kada postoji neka funkcija koja vraća anonimnu funkciju, anonimna funkcija može da vidi lokalne promenljive funkcije kojom je okružena. Međutim, nakon što je anonimna funkcija vraćena, ona može da nadživi funkciju koja ju je vratila kreirajući zatvorenje. Ovaj mehanizam dopušta pristup stanju funkcije omotaca i nadpromenljivima⁸, iako se funkcija omotac više ne izvršava. Zatvorenje je funkcija plus sve što joj je potrebno da pristupi njenim upvalues korektno. 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 koje omogućava prolazak kroz kolekciju, predstavlja se pomoću funkcije. Pri svakom pozivu ta funkcija vraća naredni element iz kolekcije. Na 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

# spoljasnja funkcija, koja se naziva fabrika, je values()
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. [4].

U kodu 14 funkcija omotac 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 itereaciji, 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()[5].

```
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:

⁸eng. 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[8]. Na listingu 16 su prikazane ove tri vrednosti.

```

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 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] Etienne Dalcol. Sailor Project, 2014-2015. on-line at: <http://sailorproject.org/main/about>.
- [2] Fengari. Fengari. on-line at: <https://fengari.io/>.
- [3] Roberto Ierusalimsky. Lua, 2003-2004. on-line at: <https://www.lua.org/start.html>.
- [4] Roberto Ierusalimsky. Lua.org, 2003-2004. on-line at: <https://www.lua.org/pil/7.1.html>.
- [5] Roberto Ierusalimsky. *Programming in Lua*. Lua.org, Rio de Janeiro, 2006.
- [6] Roberto Ierusalimsky. 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.
- [7] Kepler Project. Kepler Project, 2003. on-line at: <https://keplerproject.github.io/cgilua/>.
- [8] 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 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 na listingu 17. [8]

```
> 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 na listingu 18. [8]

```
> 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 prosledjeni iterator. Primer je dat na listingu 19. [8]

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

Kod 19: Primer funkcije `zip()`