Programski jezik Lua

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

Prvi autor, drugi autor, treći autor, četvrti autor
kontakt email prvog, drugog, trećeg, četvrtog autora

5. april 2019

Sažetak

Zahvaljujući svojoj efikasnosti, jednostavnosti i portabilnosti, Lua danas predstavlja jedan od važnijih programskih jezika i pronalazi sve češću primenu u industriji - pre svega u "embedded" sistemima. Pošto se teži tome da Lua bude mali jezik, koncepti koji se koriste u okviru njega moraju biti veoma efikasni. Neki od tih koncepata, poput tabela, metatabela, iteratora, zatvorenja i funkcija prvog reda, prikazani su u okviru ovog rada. Takođe, prikazano je i na koji način Lua može da se koristi u okviru različitih paradigmi. Iako je prvenstveno proceduralni jezik, Lua podržava mehanizme koji omogućuju pisanje u objektno-orjentisanom i funkcionalnom stilu.

Sadržaj

1 Uvod

Kada budete predavali seminarski rad, imenujete datoteke tako da sadrže redni broj teme, temu seminarskog rada, kao i prezimena članova grupe. Precizna uputstva na temu imenovnja će biti data na formi za predaju seminarskog rada. Predaja seminarskih radova biće isključivo preko veb forme, a NE slanjem mejla. Link na formu će biti dat u okviru obaveštenja na strani kursa. Vodite računa da prilikom predavanja seminarskog rada predate samo one fajlove koji su neophodni za ponovno generisanje pdf datoteke. To znači da pomoćne fajlove, kao što su .log, .out, .blg, .toc, .aux i slično, **ne treba predavati**.

2 Nastanak i istorijski razvoj, mesto u razvojnom stablu, uticaji drugih programskih jezika

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 (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 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. Operator tačka zarez je odlučeno da bude opcion.

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. Lua spada u dinamički tipizirane, "embeddable" (klasu jezika koji se često koriste da prošire ili utiču na ponašanje postojeće aplikacije) skript jezike. Podržava proceduralno, objektno orjektisano, funkcionalno i programiranje okrenuto ka podacima. Na razvojnom stablu se nalazi kao naslednik Perl-a i Pascal-a. [7]

3 Osnovna namena i mogućnosti programskog jezika Lua

Iako je Lua prvenstveno razvijena za potrebe dva projekta, danas se, zbog svoje jednostavnosti, efikasnosti i portabilnosti, koristi u najrazličitijim oblastima: ugradnim sistemima, mobilnim uređajima, veb serverima i igricama.

Lua se obično koristi 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 [4].

Ako se Lua upotrebljava u aplikaciji kao ugrađeni jezik, za njeno konfigurisanje u skladu sa domenom date aplikacije potrebno je koristiti Lua-C API. Pomoću njega se mogu, na primer, registrovati nove funkcije, praviti novi tipovi i vršiti izmene u ponašanja nekih operacija. Jedan od primera gde se Lua upotrebljava kao ugrađeni jezik je CGILua, alat za pravljenje dinamičkih veb stranica i manipulisanje podacima prikupljenih putem veb formi. U suštini, CGILua predstavlja apstrakciju za Veb server [6]. Kao ugrađeni jezik, Lua je našla i široku primenu u igricama.

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 Lue jeste u okviru programa pisanih u C-u. Lua se tada importuje kao C biblioteka. Programeri koji se opredele za ovakav način rada uglavnom najveći deo programa pišu u C-u, ali moraju dobro da poznaju i Luu kako bi kreirali jednostavne interfejse lake za upotrebu.

Lua je i tzv. jezik za spajanje (eng. glue language). Ona podržava razvoj softvera zasnovan na komponentama. U tom slučaju, aplikacija se kreira spajanjem 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. Takve komponente se spajaju pomoću Lue. Dakle, Lua se koristi za pisanje onih delova koda koji će verovatno biti menjani mnogo puta i na taj način ubrzava proces razvoja programa [4].

Kao i mnogi drugi jezici, Lua teži tome da bude fleksibilna. Ali, takođe teži tome da bude mali jezik. Za ugrađne jezike ovo je veoma bitna osobina pošto se veoma 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 tabele, funkcije prvog reda, korutine i refleksivne mogućnosti. Takođe, da bi jezik bio što manji, umesto hijerarhije numeričkih tipova (realni, racionalni, celi), Lua podržava samo brojeve u pokretnom zarezu [5].

4 Najpoznatija okruženja (framework) za korišćenje ovog jezika i njihove karakteristike

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. Međutim, veliki problem predstavlja manjak pratećih biblioteka zbog kojih je PHP dostigao toliku popularnost. Među najpopuilarnija Lua okruženja spadaju Lapis, Sailor, Luvit i Fengari.

Lapis je okruženje za pisanje veb aplikacija korišćenjem Lue ili Moonscripta (programski jezik sa lepšom i kraćom sintaksom koji se kompajlira u Luu) i izvršava se u okviru distribucije Nginx-a zvane OpenResty. OpenResty izvršava Lua (odnosno MoonScript) koristeći LuaJIT. Lapis je po svojoj funkcionalnosti analogan Rails okruženju za 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 OpenResty, već ga je moguće izvršavati i na Apache2, Nginx, Mongoose, Lighttpd, Xavante i Lwan veb serverima. Naziv je inspirisan popularnom animiranom serijom zvanom Sailor Moon, naime kada je autorka okruženja počela da uči Luu, rešila je da ako ikada napiše projekat u njoj, zvaće se Sailor jer Lua znači mesec na portugalskom. 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. 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, 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].

5 Instalacija i uputstvo za pokretanje na Linux/Windows operativnim sistemima

6 Paradigme i koncepti

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 laku integraciju između njih. Prema tome, većina programa napisanih u Lui su proceduralni, s tim što uključuju i korisne tehnike iz drugih paradigmi [5].

Tabele i objekti

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

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

Tabele

Tabele su dinamički kreirani asocijativni nizovi [1]. 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, uključujuci i nil, može biti vrednost. Tabele se skladište po referenci, a ne po vrednosti, i program jedino preko referenci (ili pokazivača) njima manipuliše. Nema skrivenih kopija ili kreiranja novih tabela iza scene.

Kreiranje tabela

Tabela se kreira uz pomoć konstruktora, koji u svojoj najjednostavnijoj formi izgleda ovako . 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 smešta njenu referencu u promenljivu a
```

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

Skladištenje vrednosti

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

`table[key] = value`

Naredni primer 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"])  -- > 10
k = 20
print(a[k])     -- > "great" - vraca vrednost (izvadi vrednost iz tabele)
a["x"] = a["x"] + 1   -- uvecava ulaz "x"
print(a["x"])  -- > 11
```

Listing 1: Primer čuvanja vrednosti u tabeli

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. Ključ tabele moze biti bilo kog tipa (pa čak i druga tabela), osim nil, i svaka tabela može da čuva vrednosti različitih tipova indeksa. Razmotrimo sledeći primer:

```
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
```

Listing 2: Primer čuvanja vrednosti u tabeli

U poslednjoj liniji primera 2 je prikazano da polja tabele dobijaju vrednost nil ako nisu inicijalizovana (ako se ne dodeli vrednost ključu u tabeli, podrazumevana (defalut) vrednost je nil) baš kao i globalne promenljive. Takođe kao kod globalnih promenljivih, polju tabele se može dodeliti vrednost nil ako se želi ????? TODO ????? 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.name kao lepši zapis za ["name"]. Dakle, poslednje tri linije prethodnog koda (primera 2) mogu se napisati i ovako:

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

Listing 3: Primer čuvanja vrednosti u tabeli

Meta tabele i meta metodi

Meta tabela je standardna tabela u Lua-i 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 ima __add polje. Ako Lua pronadje to polje, zove odgovarajuću funkciju da bi izračunala sumu. Da bi se postavila ili promenila meta tabela bilo koje tabele koristi se funkcija setmetatable, kao što je prikazano u primeru 4.

```
1000 t1 = {}
     setmetatable(t, t1)
1002 assert(getmetatable(t) == t1)
```

Listing 4: Postavljanje metatabele

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); tabela može biti svoja sopstvena meta tabela (tako da opisuje svoje individualno ponašanje). Bilo koja od ovih konfiguracija je validna.

Kreiranje meta tabele

Za kreiranje meta tabele neophodno je prvo kreirati običnu tabelu nazvanu meta. Ova tabela će imati funkciju koja se zove __add[__add je rezervisano ime za funkciju]. __add funkcija će primati dva argumenta. Levi argument će biti tabela sa poljem koje se zove value, a desni argument će biti broj:

```
1000 meta = { } -- kreira tabelu
meta.__add = function(left, right) -- dodaje meta metod
1002 return left.value + right -- pretpostavlja se da je left tabela
end
```

Listing 5: Metatabela

Dalje, treba napraviti tabelu zvanu container. Container tabela će imati promenljivu zvanu value, koja će imati vrednost 5:

```
1000 container = {
value = 5
1002 }
```

Listing 6: Metatabela

Ako se pokuša sa dodavanjem broja 4 container tabeli, Lua izbacuje sintaksnu grešku. Ovo se dešava zato što nije moguće tabeli dodati broj. Kod koji uzrokuje grešku prikazan je u primeru 7.

```
1000 result = container + 4 -- ERROR
print ("result: " .. result)
```

Listing 7: Metatabela

Dodavanjem meta tabele container tabeli, koja ima __add meta metod, može se napraviti da ovaj kod radi. Funkcija setmetatable se koristi da tabeli dodeli meta tabelu. Kod koji omogućava da sve ovo radi dat je u primeru 8.

```
1000 setmetatable(container, meta) -- postavlja meta tabelu
result = container + 4 -- RADI!
1002 print ("result: " .. result)
```

Listing 8: Metatabela

DOPUNITI... Meta tabele su možda najmoćnija karakteristika Lua-e.

Iteratori

Iterator, konstrukcija koje omogćava prolazak kroz kolekciju, predstavlja se pomoću funkcije. Pri svakom pozivu ta funkcija vraća naredni element iz kolekcije. Na listingu 9 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
```

Listing 9: 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 (eng. closure). Zatvorenje je funkcija koja se nalazi unutar neke druge funkcije. Pri tome, unutrašnja funkcija može da pristupa promenljivama spoljašnje funkcije. Ta spoljašnja funkcija, koja se zove fabrika (eng. factory), zapravo pravi zatvorenje [3].

U listingu 9 fabrika 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 je moguće isti iterator iskoristiti u više petlji bez potrebe za pravljenjem novih 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 listingu 10 prikazan je jedan iterator bez stanja - ipairs()[4].

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

Listing 10: Primer iteratora bez stanja

Rezultat izvrsavanja 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[8]. Na listingu 11 su prikazane ove tri vrednosti.

```
gen, param, state = ipairs(a)
-- rezultat izvrsavanja: function: 0x41b9e0 table: 0x1e8efb0  0
print(gen(param, state))
-- rezultat izvrsavanja: 1  one
```

Listing 11: Primer iteratora bez stanja (nastavak)

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) 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.

Kada govorimo o imenu funkcije, na primer *print*, mi zapravo pričamo o promenljivoj koja sadrži tu funkciju. Neke od posledica ove karakteristike Lua-e prikazene su u primeru 14:

```
1000    a = {p = print}
        a.p("Hello World") --> Hello World
1002    print = math.sin   -- print sada referise na funkciju sin
        a.p(print(1))      --> 0.841470
1004    sin = a.p          -- sin sada referise na funkciju print
      sin(10, 20)        --> 10      20
```

Listing 12: Funkcije

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

```
1000    function foo (x) return 2*x end
```

Listing 13: Deklaracija funkcije

Deklaracija funkcije započinje ključnom rečju function, nakon koje sledi ime funkcije, a zatim lista parametara funkcije. [2], 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:

```
1000    foo = function (x) return 2*x end
```

Listing 14: Dodela vrednosti tipa function promenljivoj foo

To jest, definicija funkcije je u stvari naredba koja promenljivoj dodeljuje vrednost tipa “function”. Izraz sa desne strane operatora dodele možemo posmatrati kao konstruktor za funkcije, baš kao što je konstruktor za tabelu. Rezultat takvog konstruktora funkcije zovemo anonimna funkcija. Iako uglavnom funkcijama dodeljujemo imena, postoje i situacije kada funkcije treba da ostanu anonimne. Na primer, biblioteka za tabele pruza 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 vrse sortiranja, sort pruza

[2]Pravilno je termin parametar funkcije koristiti za promenljivu koja čini deklaraciju funkcije, a termin argument funkcije za izraz naveden u pozivu funkcije na mestu parametra funkcije.??????????

jedan (jedini) opcioni parametar, koji predstavlja funkciju za poređenje (engl. order function, koja vrši poređenje dve vrednosti): funkcija koja 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:

```
1000     network = {
           {name = "grauna",  IP = "210.26.30.34"},
1002       {name = "arraial", IP = "210.26.30.23"},
           {name = "lua",     IP = "210.26.23.12"},
1004       {name = "derain",  IP = "210.26.23.20"},
         }
```

Listing 15: Primer tabele

Ako želimo da sortiramo tabelu po imenu polja, u obrnutom alfabetskom poretku, pišemo:

```
1000     table.sort(network, function (a,b)
           return (a.name > b.name)
1002     end)
```

Listing 16: Sortiranje tabele

Funkcija koja prima drugu funkciju kao argument, kao što je *sort*, je ono sto zovemo 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-e da upravlja funkcijama kao vrednostima prve klase.

Zatvorenja

U Lua-i, promenljive koje su lokalne za neku funkciju su takođe dostupne u funkcijama koje su definisane unutar te funkcije, tj. unutar ugnježdenih definicija. Primer koji ovo oslikava jeste kada postoji neka funkcija koja vraća anonimnu funkciju. Anonimna funkcija može da "vidi"lokalne promenljive funkcije kojom je okruzena. 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. To zatvorenje obuhvata njeno spoljašnje stanje. Ovaj mehanizam dopušta pristup stanju spoljašnje funkcije, iako se ta funkcija više ne izvršava. Ovo je prikazano u kodu 17:

```
1000 function newCounter ()
       local i = 0 -- lokalna promenljiva za newCounter() funkciju
1002   return function ()   -- anonimna funkcija

1004 -- Posto je anonimna funkcija kreirana unutar funkcije newCounter
       () ona moze da vidi
     -- sve clanove funkcije newCounter ()
1006 -- ova funkcija ce moci da zapamti stanje funkcije newCounter (),
       praveci zatvorenje

1008             i = i + 1
                 return i
1010           end
     end
1012
     c1 = newCounter()
1014 print(c1())  --> 1
     print(c1())  --> 2
```

Listing 17: Primer zatvorenja

Ovde anonimna funkcija koristi upvalue (upvalue je skraćenica za external local variable), i, da održi svoj brojač. Međutim, kada pozovemo anonimnu funkciju, *i* je izvan dometa (eng. *scope*), zato što je funkcija koja je kreirala tu promenljivu (*newCounter*) završila sa radom. Ipak, Lua razrešava tu situaciju korektno, koristeći koncept zatvorenja. Jednostavno rečeno, zatvorenje je funkcija plus sve što joj je potrebno da pristupi njenim upvalues korektno. Ako pozovemo newCounter ponovo, napraviće novu lokalnu promenljivu *i*, pa ćemo dobiti novo zatvorenje, delujući sada nad tom novom promenljivom:

```
1000    c2 = newCounter()
        print(c2())  --> 1
1002    print(c1())  --> 3
        print(c2())  --> 2
```

Listing 18: Zatvorenja

Dakle, c1 i c2 su različita zatvorenja nad istom funkcijom i svako deluje na nezavisnu instancu lokalne promenljive i. Tehnički govoreći, ono što je vrednost u Lua-i je zatvorenje, a ne funkcija. Funkcija sama za sebe je samo prototip za zatvorenje.

Funkcionalna paradigma

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. Slede primeri nekih funkcija iz ove biblioteke.

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 na listingu 19. [8]

```
1000 > each(print, map(function(x) return 2 * x end, range(4)))
     2
1002 4
     6
1004 8
```

Listing 19: 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 20. [8]

```
1000 > each(print, filter(function(x) return x % 3 == 0 end, range
         (10)))
     3
1002 6
     9
```

Listing 20: 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 21. [8]

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

Listing 21: Primer funkcije zip()

Objektno orjentisana paradigma

Objaktno orjentisana paradigma je metodologija koja ujedinjuje podatke (promenljive) i logiku (funkcije) u jednu kohezivnu jedinicu (objekat). Iako Lua nije objektno-orjentisan jezik, on nam obezbedjuje sve objekte (it does provide all the facilities) koji nam dopuštaju da implementiramo sistem objekata. Umesto klasa, u Lua-i pametna upotreba meta tabela moze da kreira sistem klasa. Meta tabele mogu da kreiraju a objekte bazirane na prototipovima.

Objekti

Objekat može biti predstavljen tabelom: promenljive instanci su regularna polja tabele, a metodi su polja tabele koja sadrže funkcije. Posebno, tabele imaju identitet (eng. selfness) koji je nezavisan od njihovih vrednosti. To znači, dva objekta (tabele) sa istom vrednošću su različiti objekti, pošto jedan objekat može imati različite vrednosti u različitim trenucima, ali je to uvek isti objekat. Kao i objekti, tabele imaju životni ciklus koji je nezavisan od toga ko ih je kreirao ili gde su kreirane. Objekti imaju svoje sopstvene operacije. Tabele takođe mogu imati operacije:

```
Account = {balance = 0}
function Account.withdraw (v)
  Account.balance = Account.balance - v
end
```

Listing 22: OOP

Ova definicija pravi novu funkciju i smesta je u polje withdraw objekta Account. Zatim, možemo je pozvati ovako:

```
Account.withdraw(100.00)
```

Listing 23: OOP

Ovakve funkcije se zovu metodi. Međutim, upotreba globalnog imena Account unutar funkcije je losa praksa. Prvo, ova funkcija ce raditi samo za ovaj (poseban) objekat. Drugo, cak i za taj specijalni (poseban) objekat funkcija ce raditi onoliko dugo koliko je objekat smesten u toj globalnoj promenljivoj. Ako promenimo ime objektu, withdraw vise nece raditi:

```
a = Account; Account = nil
a.withdraw(100.00)    -- ERROR!
```

Listing 24: OOP

7 Zaključak

Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak. Ovde pišem zaključak.

Literatura

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

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

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

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

[5] 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.

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

[7] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. The Evolution of Lua. *Proceedings of ACM HOPL III*, 2007. on-line at: https://www.lua.org/doc/hopl.pdf.

[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

Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe.