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

28. mart 2019

Sažetak

U ovom tekstu je ukratko prikazana osnovna forma seminarskog rada. Obratite pažnju da je pored ove .pdf datoteke, u prilogu i odgovarajuća .tex datoteka, kao i .bib datoteka korišćena za generisanje literature. Na prvoj strani seminarskog rada su naslov, apstrakt i sadržaj, i to sve mora da stane na prvu stranu! Kako bi Vaš seminarski zadovoljio standarde i očekivanja, koristite uputstva i materijale sa predavanja na temu pisanja seminarskih radova. Ovo je samo šablon koji se odnosi na fizički izgled seminarskog rada (šablon koji *morate* da koristite!) kao i par tehničkih pomoćnih uputstava.

Sadržaj

1 Uvod **3**

2 Nastanak i istorijski razvoj, mesto u razvojnom stablu, uticaji drugih programskih jezika **3**

3 Osnovna namena i mogućnosti programskog jezika Lua **3**

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

5 Instalacija i uputstvo za pokretanje na Linux/Windows operativnim sistemima **4**

6 Paradigme i koncepti **4**
6.1 Tabele i objekti 4
6.1.1 Tabele 4
6.1.2 Kreiranje tabela 5
6.1.3 Skladištenje vrednosti 5
6.2 Meta tabele i meta metodi 6
6.2.1 Kreiranje meta tabele 6
6.3 Iteratori 7
6.4 Funkcije 8
6.4.1 Zatvorenja 9
6.5 Funkcionalna paradigma 10
6.6 Objektno orjentisana paradigma 11
6.6.1 Objekti 11

7 Zaključak **11**

Literatura **11**

A Dodatak **12**

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

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[2].

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[4]. 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 se verovatno menjati mnogo puta i na taj način ubrzava proces razvoja programa.

Kao i mnogi drugi jezici, Lua teži tome da bude fleksibilna. Ali, takođe teži tome da bude mali jezik - i u pogledu implementacije i u terminima specifikacije. 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, na primer, tabele (koje su u suštini asocijativni nizovi), funkcije prvog reda, korutine i refleksivne mogućnosti. Takođe, da bi jezik bio što manji, umesto hijerarhije numeričkih tipova (realni, racionalni, celi), Lua ima samo double floating-point (prevesti?) tip vrednosti[3].

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

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. [3]

6.1 Tabele i objekti

Jedina struktura podataka koju Lua pruža je tabela. Tabele su dovoljno moćne da pomoću njih može da se implementira druga struktura podataka, kao sto su liste, redovi ili stekovi. Lua nije objektno-orjentisani jezik, te nema podršku za objekte. Medjutim, koristeći tabele i metatabele, sistem objekata i klasa može biti implementiran iz temelja.

6.1.1 Tabele

Tabele su first-class, dinamički kreirani asocijativni nizovi [An associative array is an array that can be indexed not only with numbers, but also with strings or any other value of the language, except nil.]. 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 nenumeričke 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.

6.1.2 Kreiranje tabela

Tabela se kreira putem konstruktora, koji u svojoj najjednostavnijoj formi izgleda ovako . Nakon što je tabela kreirana, ona treba biti dodeljena promenljivoj. Ako se tabela ne dodeli promenljivoj, nije moguće na nju referisati. Sledeći kod kreira novu tabelu i dodeljuje je promenljivoj a:

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

6.1.3 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 vratiti (izvaditi) tu vrednost:

```
k = "x"
a[k] = 10        -- nov ulaz, sa ključem key="x" i vrednošću value=10
a[20] = "great"  -- nov ulaz, sa ključem key=20 i vrednošću value="great"
print(a["x"])    -- > 10
k = 20
print(a[k])      -- > "great" - vraća vrednost (izvadi vrednost iz tabele)
a["x"] = a["x"] + 1     -- uvećava ulaz "x"
print(a["x"])    -- > 11
```

Kada u programu ne postoji ni jedna referenca na neku tabelu, Lua menadzer (upravljač) memorije će (eventualno) obrisati tu tabelu (i osloboditi memoriju koju je zauzimala), tako da memorija koju je ta tabela zauzimala kasnije može biti ponovo upotrebljena. Ključ tabele moze biti bilo kog tipa (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
```

Primetite poslednju liniju: Baš kao globalne promenljive, polja tabele dobijaju vrednost nil ako nisu inicijalizovana (ako ne dodelite vrednost ključu u tabeli, podrazumevana (defalut) vrednost je nil). Takođe kao kod globalnih promenljivih, polju tabele možete dodeliti vrednost nil ako želite da ga obrišete. 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) mogu se napisati i ovako:

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

6.2 Meta tabele i meta metodi

Meta tabele menjaju ponašanje tabela koristeći meta metode. Te meta metode su funkcije sa specifičnim imenom koje utiče na to kako se tabela ponaša. Na primer, koristeći meta tabele, 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:

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

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.

6.2.1 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:

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

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

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

Pokušajte da dodate broj 4 container tabeli; Lua će izbaciti sintaksnu grešku. Ovo je zato što ne možete da dodate broj tabeli. Kod koji uzrokuje grešku izgleda ovako:

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

Dodavanjem meta tabele container tabeli, koja ima __add meta metod, mozemo napraviti da ovaj kod radi. Funkcija setmetatable se koristi da dodeli meta tabelu. Kod koji omogućava da sve ovo radi izgleda ovako:

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

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

6.3 Iteratori

Iterator, konstrukcija koje omogćava prolazak kroz kolekciju, predstavlja se pomoću funkcije. Pri svakom pozivu ta funkcija vraća naredni element iz kolekcije. Na listingu 1 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 1: 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 [1].

U listingu 1 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 možemo isti iterator iskoristiti u više petlji bez potrebe da pravimo nova zatvorenja. U svakoj itreaciji, for petlja poziva svoj iterator sa dva argumenta: invarijantnim stanjem i kontrolnom promenljivom. Iterator bez stanja generiše naredni element na osnovu te dve vrednosti. U listingu 2 prikazan je jedan iterator bez stanja - ipairs()[2].

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

Listing 2: 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[5]. Na listingu 3 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 3: Primer iteratora bez stanja (nastavak)

6.4 Funkcije

Funkcije u Lua-i su first-class values (primarne vrednosti, građani prvog reda) sa odgovarajućim "leksičkim opsegom". Za funkciju se kaže da je građanin prvog reda (vrednosti 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 se vrate kao povratna vrednost 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 (čuva) tu funkciju. Neke od posledica ove karakteristike Lua-e prikazene su u narednom primeru:

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

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

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

Deklaracija funkcije započinje ključnom rečju function, nakon koje sledi ime funkcije, a zatim lista parametara funkcije. [Parametri su imena promenljivih okružena zagradama], 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
```

To jest, definicija funkcije je u stvari naredba koja promenljivoj dodeljuje vrednost tipa “function”. Izraz function (x) ... end mozemo 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 (nekoliko) situacija kada funkcije treba da ostanu anonimne. Na primer, biblioteka za tabele pruza funkciju table.sort, koja prima tabelu i sortira njene elemente. Ova funkcija mora da dozvoli (omogući) neograničeno varijacija in the sort order: rastuće ili opadajuće, numeričko ili alfabetsko, tabele sortirane po ključu, itd. Umesto kucanja posebnog koda da bile omogućene sve vrste opcija (načina sortiranja), sort pruza 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:

```
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"},
}
```

Ako želimo da sortiramo tabelu po imenu polja, u obrnutom alfabetskom poretku (in reverse alphabetical order), pišemo:

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

Funkcija koja prima drugu funkciju kao argument, kao što je sort, je ono sto zovemo funkcija višeg reda (a higher-order function). Higher-order funkcije su moćan mehanizam u programiranju I korišćenje anonimnih funkcija za kreiranje argumenata funkcije je odličan izvor fleksibilnosti. Ali funkcije višeg reda nemaju specijalna prava; have no special rights; one su samo posledica mogućnosti Lua-e da upravlja funkcijama kao prvoklasnim vrednostima.

6.4.1 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 (enclosing) funkcije kojom je okruzena. Međutim, pošto je anonimna funkcija vraćena, ona može da nadživi postojeću funkciju. Kada vratimo anonimnu funkciju, ona kreira zatvorenje. To zatvorenje obuhvata njeno enclosing stanje (visible chunks). Ovaj mehanizam dopušta pristup stanju enclosing funkcije, iako se ta funkcija više ne izvršava. Ovo je prikazano u sledećem kodu:

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

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

              i = i + 1
              return i
            end
    end

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

Ovde anonimna funkcija koristi upvalue (upvalue je skraćenica za external local variable), i, da održi svoj brojač. Međutim, kada pozovemo anonimnu funkciju, i je van dometa (izvan scope-a), zato što je funkcija koja je kreirala tu promenljivu (newCounter) završila sa radom (vratila vrednost, returned). 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:

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

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

6.5 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 4. [5]

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

Listing 4: 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 5. [5]

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

Listing 5: 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 6. [5]

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

Listing 6: Primer funkcije zip()

6.6 Objektno orjentisana paradigma

OOP 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.

6.6.1 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
```

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

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

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!
```

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

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

[3] 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.

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

[5] 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.