UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Milena Dukanac

SVOJSTVA JEZIKA ELIXIR KROZ PRIMENE U OBLASTI SKLAPANJA GENOMA

master rad

Beograd, 2015.

Mentor:

dr Milena Vujošević Janičić, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Jovana Kovačević, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

dr Vesna Marinković, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami i tati

Naslov master rada: Svojstva jezika Elixir kroz primene u oblasti sklapanja genoma

Rezime:

Ključne reči: analiza, geometrija, algebra, logika, računarstvo, astronomija

Sadržaj

Glava 1

Uvod

1.1 Elixir

Erlang

Godine 1981. oformljena je nova laboratorija, Erikson CSLab (eng. The Ericsson CSLab) u okviru firme Erikson sa ciljem da predlaže i stvara nove arhitekture, koncepte i strukture za buduće softverske sisteme [6]. Eksperimentisanje sa dodavanjem konkurentnih procesa u programski jezik Prolog je bio jedan od projekata Erikson CSLab-a i predstavlja začetak novog programskog jezika. Taj programski jezik je 1987. godine nazvan Erlang (fusnotu dodaj). Sve do 1990, Erlang se mogao posmatrati kao dijalekt Prologa. Od tada, Erlang ima svoju sintaksu i postoji kao potpuno samostalan programski jezik. Godine rada su rezultirale u sve bržim, boljim i stabilnijim verzijama jezika, kao i u nastanku standardne biblioteke OTP (eng. The Open Telecom Platform) [6]. Od decembra 1998. godine, Erlang i OTP su postali deo slobodnog softvera (eng. open source software) i mogu se slobodno preuzeti sa Erlangovog zvaničnog sajta [9]. Danas, veliki broj kompanija koristi Erlang u razvoju svojih softverskih rešenja. Neke od njih su: Erikson, Motorola, Votsap (eng. WhatsApp), Jahu (eng. Yahoo!), Fejsbuk (eng. Facebook) [9].

Erlang je funkcionalan, deklarativan i konkurentan programski jezik. Na njega, kao na funkcionalan jezik, uticao je Lisp funkcionalnom paradigmom koju je prvi predstavio. Na planu konkurentnosti Erlang je svojevrstan primer. Na početku, Erlang je stvaran kao dodatak na Prolog, vremenom prerastao u dijalekt Prologa, a kasnije je zbog svoje kompleksnosti i sveobuhvatnosti evoluirao u potpuno novi

programski jezik. Stoga je uticaj Prologa na Erlang bio neminovan. Sintaksa Erlanga u velikoj meri podseća na Prologovu (na primer, promenljive moraju počinjati velikim slovom, svaka funkcionalna celina se završava tačkom), oba jezika u velikoj meri koriste poklapanje obrazaca (eng. pattern matching).

Sa druge strane, Erlang je uticao na nastanak programskog jezika Eliksir (eng. Elixir). Eliksir, uz izmenjenu Erlangovu sintaksu, dopunjenu Erlangovu standardnu biblioteku, uživa široku popularnost.

Elixir

Tokom 2010. godine *Jose Valim*, u to vreme zaposlen na poziciji programera u kompaniji *Plataformatec*, radio je na poboljšanju performansi *Ruby on Rails* framework-a na višejezgarnim sistemima. [6] Shvatio je da Ruby nije bio dovoljno dobro dizajniran da reši problem konkurentnosti, pa je započeo istraživanje drugih tehnologija koje bi bile prihvatljivije. Tako je otkrio Erlang, i upravo ga je interesovanje prema Erlangovoj virtuelnoj mašini podstaklo da započne pisanje jezika Elixir. Uticaj projekta na kome je do tada radio odrazio se na to da Elixir ima sintaksu koja je nalik na Ruby-jevu. Ovaj jezik se pokazao veoma dobro pri upravljanju milionima simultanih konekcija: u 2015. je zabeleženo upravljanje nad 2 miliona WebSocket konekcija [4], dok je u 2017. za skalirani Elixir zabeležena obrada 5 miliona istovremenih korisnika. Elixir se danas koristi u velikim kompanijama, kao što su Discord [1] i Pinterest[5].

Elixir je dinamički tipiziran, funkcionalni programski jezik koji se pokreće na vituelnoj mašini jezika Erlang, pa samim tim i nasleđuje pogodna svojstva kao što su konkurentnost i tolerisanje grešaka, koje dolaze sa ovim okruženjem.[11] Iz tačke gledišta olakšavanja svakodnevnog razvoja sofvera, koncepti kao što su metaprogramiranje - tehnika kojom programi imaju mogućnost da druge programe posmatraju kao svoje podatke i na taj način čitaju, pa čak i modifikuju njihov, a samim tim i svoj kod u vreme izvršavanja, zatim polimorfizam i makroi, kao i podrška za alate su falili Erlangovom ekosistemu i upravo je Elixir to nadomestio. Cilj ovog rada je da čitaoca bliže upozna sa osnovnim osobinama, funkcionalnostima i specifičnostima ovog jezika kao i da kroz primere pokuša da približi programerske prakse korišćene u ovom jeziku.

Osobine jezika Elixir

U ovom delu će biti opisane osobine jezika Elixir, osnove njegove sintakse, semantike, kao i podrška za osnovne koncepte funkcionalnih jezika poput poklapanja obrazaca (eng. Pattern matching) i imutabilnosti podataka.[11][10]

Pre nego što započnemo priču o tipovima, treba pomenuti **Kernel**. To je podrazumevano okruženje koje se koristi u Elixiru. Ono sadrži primitive jezika kao što su: aritmetičke operacije, rukovanje procesima i tipovima, makroe za definisanje novih funkcionalnosti (funkcija, modula...), provere guard-ova - predefinisanog skupa funkcija i makroa koji proširuju mogućnost pattern matching-a itd.[7]

Poklapanje obrazaca

Poklapanje obrazaca je proveravanje da li se u datoj sekvenci tokena može prepoznati neki šablon. Na praktičnom primeru operatora = ovaj koncept će biti jasniji. U većini programskih jezika, operator = je operator dodele, koji levoj strani dodeljuje vrednost izraza na desnoj. U Elixiru se naziva **operator uparivanja (eng. matching)**. On se uspešno izvršava, ako pronađe način da izjednači levu stanu (svoj prvi operand) sa desnom (drugi operand).

Na primer, ako bismo pokušali da napisemo $2 + 2 = 5$, dobili bismo sledeće:

```
iex(7)> 5 = 2 + 2
** (MatchError) no match of right hand side value: 4
```

Slika 1.1:

Elixir nam ovde kaže da $2 + 2$ zaista nije 5. U Elixir-u leva strana mora da ima istu vrednost kao i desna strana.

```
iex(7)> 4 = 2 + 2
4
```

Slika 1.2:

Slično, 2 identična stringa sa obe strane znaka jednakosti će dati sledeći rezultat:

```
iex(8)> "pas" = "pas"
"pas"
```

Slika 1.3:

Pogledajmo kako možemo prikazati poklapanje obrazaca radeći sa listama. Pretpostavimo da imamo listu osoba:

```
iex(1)> lista = ["Mika Mikic", "Pera Peric", "Ana Anic", "Mara Maric"]
["Mika Mikic", "Pera Peric", "Ana Anic", "Mara Maric"]
```

Slika 1.4:

Neka na primer hoćemo da zapamtimo prve tri osobe, a četvrta nam nije bitna. U te svrhe možemo iskoristiti poklapanje obrazaca.

```
iex(2)> [prvi,drugi,treci|ostali] = lista
["Mika Mikic", "Pera Peric", "Ana Anic", "Mara Maric"]
```

Slika 1.5:

Ovim kažemo Elixir-u da dodeli prvu, drugu i treću stavku iz liste promenljivama prvi, drugi i treći. Takođe mu kažemo da ostatak liste dodeli pomenljivoj ostali, a to navodimo pomoću **pipe operatora** (|). Možemo da proverimo šta se desilo posmatrajući vrednosti svake od ovih promenljivih.

```
iex(3)> prvi
"Mika Mikic"
iex(4)> drugi
"Pera Peric"
iex(5)> treci
"Ana Anic"
iex(6)> ostali
["Mara Maric"]
```

Slika 1.6:

Imutabilnost podataka

U mnogim programskim jezicima je dozvoljeno da dodeljujemo vrednost promenljivoj, a zatim da je menjamo tokom izvršavanja programa. Mogućnost da zamenimo vrednost na određenoj memorijskoj lokaciji drugom vredošću čini se legitimna i čini se da povećava čitljivost našeg programa. Tokom izvršavanja programa obično ne znamo tačno vreme kada se ova promena dešava i obično i ne vodimo računa o tome dok pišemo naše programe. Ali šta se dešava kada se vrednost u memoriji promeni u trenutku kada je koristi više instanci našeg programa? Pretpostavimo da se ne menja samo vrednost, već i tip. Ovakvo ponasanje je poznato kao **mutabilnost (promenljivost)**. U konkurentnim okruženjima je izvor grešaka koje je veoma teško pratiti i reprodukovati. Mutabinost takođe vodi ka veoma komplikovanom kodu, pisanom ad-hoc kako bi se rešili problemi sinhronizacije. Ovo može smanjiti rizik da konkurentni procesi pristupaju istim resursima, ali po veoma visokoj ceni.

Umesto toga, drugi jezici, kao što je Erlang, a samim tim i Elixir imaju osobinu **imutabilnosti (nepromenljivosti)**. Oni jednostavno ne dozvoljavaju promenu vredosti na određenoj memorijskoj lokaciji. Na ovaj način, ako promenljivoj a dodelimo vrednost 1, sigurno znamo da se ta vrednost neće menjati tokom izvršavanja našeg programa i da ne moramo voditi računa o problemima sinhronizacije u konkurentnom okruženju.

Osnovni tipovi

Elixir ima svoje ugrađene (primitivne) tipove. To su:

1. Atomi
2. Celi brojevi
3. Brojevi u pokretnom zarezu
4. Procesi
5. Portovi
6. Ugrađene torke
7. Liste
8. Mape

9. Funkcije
10. Niske bitova
11. Reference

Svaki od ovih tipova, osim poslednja dva, imaju odgovarajuće module koji sadrže funkcije koje se koriste za operacije nad tim tipom. Oni predstavljaju omotač oko primitivnog tipa koji nam omogućava da koristimo dodatne funkcionalnosti nad njim.

Atomi

Atomi su konstante ili simboli, gde njihovo ime predstavlja njihovu vrednost.Počinju dvotačkom (:) i mogu sadržati slova, cifre, simbole _, @. Mogu se završavati sa ! i ?. Atomi se mogu naći svuda u Elixir-u. Oni su ključevi za listu ključnih reči, koji se često koriste da označe uspeh ili grešku, npr. :ok i mnogi drugi.

Celi brojevi

Celi brojevi su slični kao i u većini ostalih jezika i mogu biti dekadni, heksadekadni, oktalni i binarni. Karakter _ se može koristiti za odvajanje blokova cifara. Veoma značajna stvar je da ne postoji fiksna veličina za čuvanje celih brojeva u memoriji, već interna reprezentacija raste kako bi broj mogao biti smešten u celosti.

Brojevi u pokretnom zarezu

Brojevi u pokretnom zarezu se u memoriji zapisuju po standardu IEEE 754, a za zapisivanje konstanti ovog tipa koristi se tačka između najmanje 2 cifre. Takođe je moguće koristiti notaciju koja obuhvata navođenje eksponenata.

Procesi

U Elixir-u se sav kod pokreće unutar procesa. Procesi su izolovani jedni od drugih, spajaju se jedan sa drugima i komuniciraju putem slanja poruka. Procesi nisu samo osnova za konkurentnost u Elixir-u, već i sredstva za izgradnju distribuiranih i tolerantnih programa. Procese Elixir-a ne treba mešati sa procesima operativnog sistema. Procesi Elixir-a su izuzetno lagani u smislu memorije i CPU-a (čak i u poređenju sa nitima koji se koriste u mnogim drugim programskim jezicima). Zbog toga nije neuobičajeno da se istovremeno odvijaju desetine ili čak stotine hiljada procesa.

Portovi i reference

Portovi ukazuju na spoljne resurse koje omogućavaju interakciju sa spoljnim svetom.

Reference su jedinstvene vrednosti u globalnom kontekstu izvršavanja programa koje se kreiraju pozivom make_ref funkcije.

Liste

Lista se čuvaju u memoriji kao povezane liste, što znači da svaki element u listi čuva svoju vrednost i ukazuje na sledeći element sve dok se ne dostigne kraj liste. To znači da je pristup proizvoljnom elementu liste kao i određivanje družine liste linearna operacija, jer je potrebno da prođemo celu listu da bismo odredili njenu dužinu. Slično, performanse spajanja dve liste zavise od dužine one koja se nalazi sa leve strane.

Elixir koristi uglaste zagrade ([]) da označi listu vrednosti. Vredosti mogu biti bilo kog tipa:

```
iex(9)> [1, 2, true, 3]
[1, 2, true, 3]
iex(10)> length([1, 2, true, 3])
4
```

Slika 1.7:

Dve liste se mogu nadovezati ili oduzeti korišćenjem operatora $++/2$ i $--/2$, redom:

```
iex(11)> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]
iex(12)> [1, false, 2, true, 3, false] -- [true, false]
[1, 2, 3, false]
```

Slika 1.8:

Operatori liste nikada ne menjaju postojeću listu. Povezivanje ili uklanjanje elemenata iz liste vraća novu listu, jer su strukture podataka u Elixir-u nepromenljive. Jedna od prednosti nepromenljivosti je to što dovodi do jasnijeg koda. Možete slobodno proslediti podatke sa garancijom da ih niko neće mutirati u memoriji - samo transformisati.

Lista može biti prazna ili se može sastojati od **glave** i **repa**. Glava je prvi element liste, a rep je ostatak liste. Oni se mogu izdvojiti pomoću funkcija *hd/1* i *tl/1*. Dodelimo listu promenljivoj i dohvatimo njenu glavu i rep:

```
iex(13)> lista = [1, 2, 3, 4]
[1, 2, 3, 4]
iex(14)> hd(lista)
1
iex(15)> tl(lista)
[2, 3, 4]
```

Slika 1.9:

Ako pokušamo da izdvojimo glavu ili rep od prazne liste, dobićemo grešku.

Nekada će se desiti da kreiramo listu i da će ona vratiti vredosti pod jednostrukim navodnicima. Na primer:

```
iex(16)> [11, 12, 13]
'\v\f\r'
iex(17)> [104, 101, 108, 108, 111]
'hello'
```

Slika 1.10:

Kada Elixir vidi listu *ASCII* brojeva za štampanje, Elixir će to ispisati kao *char* listu (doslovno listu znakova). Char liste su prilično uobičajene kada se povezuju sa postojecim Erlang kodom. Kad god vidite vrednost u iex-u i niste sasvim sigurni šta je to, možete koristiti *i/1* da biste preuzeli informacije o tome:

```
iex(18)> i 'hello'
Term
  'hello'
Data type
  List
Description
  This is a list of integers that is printed as a sequence of characters
  delimited by single quotes because all the integers in it represent valid
  ASCII characters. Conventionally, such lists of integers are referred to
  as "charlists" (more precisely, a charlist is a list of Unicode codepoints,
  and ASCII is a subset of Unicode).
Raw representation
  [104, 101, 108, 108, 111]
Reference modules
  List
Implemented protocols
  Collectable, Enumerable, IEx.Info, Inspect, List.Chars, String.Chars
```

Slika 1.11:

Torke

Treba da imamo na umu da reprezentacije sa jednostrukim i dvostrukim navodnicima nisu ekvivalentne u Elixir-u i da predstavljaju različite tipove:

```
iex(19)> 'hello' == "hello"
false
```

Slika 1.12:

Torke se u Elixir-u definišu pomoću vitičastih zagrada. Kao i liste, mogu sadržati vrednosti bilo kog tipa:

```
iex(20)> {:ok, "hello", 1}
{:ok, "hello", 1}
iex(21)> tuple_size {:ok, "hello", 1}
3
```

Slika 1.13:

Torke su strukture fiksne dužine koje bi trebalo da sadrže svega nekoliko elemenata koji su zapisani u memoriji jedan za drugim. To znači da se pristup elementu torke ili određivanje dužine torke izvršava u konstantnom vremenu. Razlika u odnosu na liste je u semantici njihove upotrebe. Liste se koriste kada se manipuliše kolekcijom, dok se torke, zbog brzine pristupa elementima torke, uglavnom koriste kako bi se u njih smestile povratne vrednosti funkcije. Indeksi počinju od nule:

```
iex(22)> tuple = {:ok, "hello", 1}
{:ok, "hello", 1}
iex(23)> elem(tuple, 1)
"hello"
```

Slika 1.14:

Takođe je moguće umetnuti novi element na odredjeno mesto u torki pomoću funkcije *put_elem*/3:

```
iex(24)> tuple = {:ok, "hello", 1}
{:ok, "hello", 1}
iex(25)> put_elem(tuple, 1 , "world")
{:ok, "world", 1}
iex(26)> tuple
{:ok, "hello", 1}
```

Slika 1.15:

Obratimo pažnju da je funkcija *put_elem*/3 vratila novu torku. Originalna torka smeštena u promenljivoj *tuple* nije izmenjena. Kao i liste, torke su takođe nepromenljive. Svaka operacija nad torkom vraća novu torku, nikada ne menja postojeću. Ova operacija, kao i operacija ažuriranja torke je skupa, jer zahteva kreiranje nove torke u memoriji.

Ovo se odnosi samo na samu torku, a ne na njen sadržaj. Na primer, kada ažurirate torku, svi unosi se dele između stare i nove torke, osim unosa koji je zamenjen. Drugim rečima, torke i liste u Elixir-u mogu da dele svoj sadržaj. Ovo smanjuje količinu memorije koju jezik treba da zauzme i moguće je samo zahvaljujući nepromenljivoj semantici jezika.

Ove karakteristike performansi diktiraju upotrebu tih struktura podataka. Kao što je već pomenuto, jedan od uobičajenih slučajeva korišćenja torki je za vraćanje dodatnih vrednosti iz funkcije. *File.read/1* je funkcija koja se može koristiti za čitanje sadržaja datoteke. Ona vraća torku:

```
iex(34)> File.read("C:\elixir\text_document.txt")
{:error, :enoent}
```

Slika 1.16:

Ako putanja do fajla postoji, povratna vrednost funkcije je torka sa prvim elementom koji je atom :ok i drugim elementom koji je sadržaj datog fajla. U suprot-

nom, povratna vrednost funkcije će biti torka gde je prvi element atom :error, a drugi element opis greške.

Liste ključnih reči i mape

Elixir podržava asocijativne strukture podataka. Asocijativne strukture podataka su one koje su u stanju da pridruže određenu vrednost ili više vrednosti ključu. Dve glavne među njima su **liste ključnih reči** i **mape**.

Liste ključnih reči

U mnogim funkcionalnim programskim jezicima, uobičajeno je da se koristi lista dvočlanih torki za predstavljanje strukture podataka ključ - vrednost. U Elixir-u, kada imamo listu torki gde je prvi element torke atom (tj. ključ), mi takvu listu nazivamo lista ključnih reči:

```
iex(49)> lista = [{:a, 1}, {:b, 2}, {:c, 3}]
[a: 1, b: 2, c: 3]
iex(50)> lista == [a: 1, b: 2, c: 3]
true
```

Slika 1.17:

Elixir podržava posebnu sintaksu za definisanje takvih lista: $[key : value]$. Zapravo, one mapiraju listu torki iz prethodnog primera. Kako su liste ključnih reči liste, nad njima možemo koristiti sve operacije dostupne nad listama. Na primer, možemo da koristimo ++ da dodamo nove vrednosti listi ključnih reči:

```
iex(51)> lista ++ [d: 4]
[a: 1, b: 2, c: 3, d: 4]
iex(52)> [a: 0] ++ lista
[a: 0, a: 1, b: 2, c: 3]
```

Slika 1.18:

Elementima liste ključnih reči se pristupa na sledeći način:

```
iex(57)> lista = [a: 0, a: 1, b: 2, c: 3]
[a: 0, a: 1, b: 2, c: 3]
iex(58)> lista[:a]
0
```

Slika 1.19:

Liste ključnih reči su važne, jer imaju tri posebne karakteristike:

1. Ključevi moraju biti atomi.
2. Ključevi su uredjeni, onako kako je navedeno od strane programera.
3. Ključevi se mogu ponavljati.

Elixir obezbeđuje modul za ključne reči kako bismo mogli manipulisati njima. Ipak, liste ključnih reči su jednostavno liste, i kao takve pružaju iste karakteristike linearnih performansi kao i liste. Što je lista duža, duže ce biti potrebno pronaci ključ, brojati stavke i tako dalje. Iz tog razloga, liste ključnih reči se koriste u Elixir-u uglavnom za prosleđivanje opcionih vrednosti. Ako želimo da sačuvamo mnogo elemenata ili da garantujemo pojavljivanje jednog ključa sa maksimalno jednom vrednošcu, treba da koristitimo mape.

Mape

Mapa je kolekcija koja sadrži parove ključ-vrednost. Glavne razlike između liste parova ključ-vrednost i mape su u tome sto mape ne dozvoljavaju ponavljanje ključeva, jer su to asocijativne strukture podataka, i sto ključevi mogu biti bilo kog tipa. Mapa je veoma efikasna struktura podataka, narocito kada količina podataka raste. Ukoliko želimo da podaci u kolekciji ostanu baš u onom redosledu u kom smo ih naveli inicijalno, onda je bolje koristiti liste parova ključ-vrednsot, jer mape ne prate nikakvo uređenje.

Mapa se definiše pomoću sintakse %{}:

```
iex(63)> mapa = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex(64)> mapa[:a]
1
iex(65)> mapa[2]
:b
iex(66)> mapa[:c]
nil
```

Slika 1.20:

Za razliku od liste ključnih reči, mape su vrlo korisne kod poklapanja obrazaca. Kada se mapa koristi u poklapanju obrazaca, ona će se uvek podudarati sa poskupom date vrednosti:

```
iex(67)> %{} = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex(68)> %{:a => a} = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex(69)> a
1
iex(70)> %{:c => c} = %{:a => 1, 1 => :b}
** (MatchError) no match of right hand side value: %{1 => :b, :a => 1}
    (stdlib) erl_eval.erl:453: :erl_eval.expr/5
    (iex) lib/iex/evaluator.ex:257: IEx.Evaluator.handle_eval/5
    (iex) lib/iex/evaluator.ex:237: IEx.Evaluator.do_eval/3
    (iex) lib/iex/evaluator.ex:215: IEx.Evaluator.eval/3
    (iex) lib/iex/evaluator.ex:103: IEx.Evaluator.loop/1
    (iex) lib/iex/evaluator.ex:27: IEx.Evaluator.init/4
```

Slika 1.21:

Kao što vidimo, mapa se poklapa sve dok ključevi u obrascu postoje u datoj mapi. Tako, prazna mapa odgovara svim mapama.

Promenljive se mogu koristiti prilikom pristupa, poklapanja i dodavanja ključeva mape:

```
iex(78)> mapa = %{a => :jedan}
%{1 => :jedan}
iex(79)> mapa[a]
:jedan
iex(80)> %{^a => :jedan} = %{1 => :jedan, 2 => :dva, 3 => :tri}
%{1 => :jedan, 2 => :dva, 3 => :tri}
```

Slika 1.22:

Modul **Map** nam obezbeđuje razne funkcije za manipulaciju mapama:

```
iex(81)> Map.get(%{:a => 1, 2 => :b}, :a)
1
iex(82)> Map.put(%{:a => 1, 2 => :b}, :c, 3)
%{2 => :b, :a => 1, :c => 3}
iex(83)> Map.to_list(%{:a => 1, 2 => :b})
[{2, :b}, {:a, 1}]
```

Slika 1.23:

Mape imaju sledeću sintaksu za azuriranje vrednosti ključa:

```
iex(84)> mapa = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex(85)> %{mapa | 2 => "dva"}
%{2 => "dva", :a => 1}
iex(86)> %{map | :c => 3}
** (CompileError) iex:86: undefined function map/0
    (stdlib) lists.erl:1354: :lists.mapfoldl/3
```

Slika 1.24:

Gore navedena sintaksa zahteva da dati ključ postoji u mapi i ne može se koristiti za dodavanje novih ključeva. Na primer, korišćenje ove sintakse za ključ :c nije uspelo, jer ključ :c ne postoji u mapi.

Ukoliko su svi ključevi u mapi atomi, onda možemo koristiti sintakcu ključnih reci radi pogodnosti:

```
iex(86)> map = %{a: 1, b: 2}
%{a: 1, b: 2}
```

Slika 1.25:

Jos jedno zanimljivo svojstvo mapa je to što obezbeđuju sopstvenu sintaksu za pristup atomskim ključevima:

```
iex(87)> mapa = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex(88)> mapa.a
1
iex(89)> mapa.c
** (KeyError) key :c not found in: %{2 => :b, :a => 1}
```

Slika 1.26:

Programeri koji programiraju u Elixir-u obično radije koriste *map.field* sintaksu i poklapanje obrazaca umesto funkcija iz modula Map, kada rade sa mapama, jer dovode do asertivnog stila programiranja.

Često ćemo imati mape unutar mapa ili čak liste ključnih reči unutar mapa. Elixir obezbeđuje pogodnosti za manipulisanje ugnježdenim strukturama podataka poput $put\_in/2$, $update\_in/2$ i drugih naredbi koje daju iste pogodnosti koje bismo mogli pronaći u imperativnim jezicima, a da pritom zadrže nepromenljiva svojstva jezika.

Zamislite da imamo sledeću strukturu:

```
iex(89)> users = [john: %{name: "John", age: 27, languages: ["Erlang", "Ruby", "
Elixir"]}, mary: %{name: "Mary", age: 29, languages: ["Elixir", "F#", "Clojure"]
}]
[
  john: %{age: 27, languages: ["Erlang", "Ruby", "Elixir"], name: "John"},
  mary: %{age: 29, languages: ["Elixir", "F#", "Clojure"], name: "Mary"}
]
```

Slika 1.27:

Imamo listu ključnih reči korisnika gde je svaka vrednost mapa koja sadrži ime, starost i listu programskih jezika koje svaki korisnik voli. Ako hoćemo da pristupimo godinama od Džona, to bismo mogli uraditi ovako:

```
iex(90)> users[:john].age
27
```

Slika 1.28:

```
iex(91)> users = put_in users[:john].age, 31
[
  john: %{age: 31, languages: ["Erlang", "Ruby", "Elixir"], name: "John"},
  mary: %{age: 29, languages: ["Elixir", "F#", "Clojure"], name: "Mary"}
]
```

Slika 1.29:

Istu sintaksu možemo koristiti za ažuriranje vrednosti:

Makro *update_in*/2 je sličan, ali nam omogućava da prosledimo funkciju koja kontorliše kako se vrednost menja. Na primer, ako želimo da uklonimo programski jezik Clojure sa Marijinog spiska jezika, to možemo uraditi ovako:

```
iex(92)> users = update_in users[:mary].languages, fn languages -> List.delete(l
anguages, "Clojure") end
[
  john: %{age: 31, languages: ["Erlang", "Ruby", "Elixir"], name: "John"},
  mary: %{age: 29, languages: ["Elixir", "F#"], name: "Mary"}
]
```

Slika 1.30:

Postoji i funkcija *get_and_update_in* koja nam omogućava da izvučemo vrednost i ažuriramo strukturu podataka odjednom. Tu su i funkcije *put_in*/3, *update_in*/3 i *get_and_update_in*/3 koje nam omogućavaju dinamički pristup strukturama podataka.

Osnovni operatori

Pored osnovnih artimetičkih operatora +, -, *, /, kao i funkcija div/2 i rem/2 za celobrojno deljenje i ostatak pri celobrojnom deljenju, Elixir podržava i već pomenute operatore ++ i −− za nadovezivanje i oduzimanje listi. Zatim, operator koji se koristi za nadovezivanje stringova je operator <>.

Elixir takođe obezbeđuje 3 bool operatora: or, and i not. Oni su striktni u smislu da očekuju nesto što ima vrednost true ili false kao svoj prvi operand:

```
iex(39)> true and true
true
iex(40)> false or is_atom(:example)
true
```

Slika 1.31:

Ukoliko kao prvi operand prosledimo nesto čija vrednost nije tipa bool, dobićemo grešku:

```
iex(42)> 1 and true
** (BadBooleanError) expected a boolean on left-side of "and", got: 1
```

Slika 1.32:

Or i and su lenji operatori, jer desni operand izračunavaju, samo u slučaju da levi nije dovoljan za određivanje rezultata.

Pored ovih boolean operatora, Elixir takođe obezbeđuje ||, i ! koji prihvataju argumente bilo koje vrste. Za ove operatore, sve vrednosti osim false i nil ce se proceniti na true:

Možemo smatrati pravilom da kada očekujemo bool vrednsti, koristimo and i or, a ako bilo koji od operanada ima vrednosti koji nisu tipa bool, onda koristimo , || i !.

Elikir takođe obezbeđuje ==,! =, ===,! ==, <=,> =, <i> kao operatore poređenja, pri čemu se operator === od operatora == razlikuje po tome sto pored vrednosti, poredi i tip.

Možemo porediti i tipove među sobom. Razlog zbog kojeg možemo uporediti različite tipove podataka je pragmatizam. Algoritmi sortiranja ne moraju da brinu

```
iex(42)> 1 || true
1
iex(43)> false || 11
11
iex(44)> nil && 13
nil
iex(45)> true && 17
17
iex(46)> !true
false
iex(47)> !1
false
iex(48)> !nil
true
```

Slika 1.33:

o različitim tipovima podataka da bi se sortirali. Ukupan redosled sortiranja je definisan u nastavku:

number < atom < reference < function < port < pid < tuple < map < list < bitstring

Odlučivanje

Strukture odlučivanja zahtevaju da programer odredi jedan ili više uslova koje će program proceniti ili testirati zajedno sa naredbom ili naredbama koje treba izvršiti, ako je uslov određen ili tačan, i opciono, druge naredbe koje treba izvršiti, ako je utvrđeno da je uslov netačan.

Elixir obezbedjuje **if/else** uslovne konstrukte kao i mnogi drugi programski jezici. On takođe ima naredbu cond koja poziva prvu tačnu vrednost koju pronađe.

Case je još jedan kontrolni tok koji koristi poklapanje obrazaca za kontrolu toka programa.

Elixir ima sledeće vrste naredbi za odlučivanje:

1. if naredba - If naredba se satoji od bool izraza praćenog ključnom reči *do*, jedne ili više izvršnih naredbi i na kraju ključne reči. *end*

2. if..else naredba - If naredba može biti praćena naredbom else (unutar do..end bloka), koja se izvršava, ako je bool izraz netačan.

3. unless naredba - Naredba unless ima isto telo kao i if naredba. Kod unutar unless naredbe se izvršava samo kada je navedeni uslov netačan.

4. unless..else - Naredba unless...else ima isto telo kao i naredba if..else. Kod unutar unless..else naredbe se izvršava samo kada je navedeni uslov netačan.

5. cond - Naredba cond se koristi kada želimo da izvršimo neki kod na osnovu nekoliko uslova. Radi kao if..else if..else kod drugih programskih jezika.

6. case - Naredba case se može smatrati zamenom za **switch** naredbu u imperativnim programskim jezicima. Naredba case uzima promenljivu ili literal i primenjuje odgovarajući obrazac poklapanja u različitim slučajevima. Ako se bilo koji slučaj poklapa, Elixir izvršava kod povezan sa tim slučajem i izlazi iz case naredbe.

Moduli

U Elixir-u možemo grupisati nekoliko funkcija u module. Već smo pominjali različite module u prethodnim odeljcima (Map, Enum, List, String,...). Da bismo kreirali sopstvene module u Elixir-u, koristimo makro **defmodule**, a da bismo definisali svoje funkcije, koristimo makro **def**:

```
1  defmodule Math do
2    def sum(a, b) do
3      a + b
4    end
5  end
```

Slika 1.34:

Moduli mogu biti ugnježdeni u Elixir-u. Ova osobina jezika nam omogućava da svoj kod organizujemo na što bolji način. Da bismo ugnjezdili module, koristimo sledeću sintaksu:

```
1   defmodule Math do
2     defmodule Adding do
3       def sum(a, b) do
4         a + b
5       end
6     end
7   end
```

Slika 1.35:

Gore navedeni primer definiše 2 modula: Math i Math.Adding. Drugom se može pristupati samo pomoću Adding unutar Math modula sve dok su u istom leksičkom opsegu. Ako se kasnije Adding modul premesti izvan definicije Math modula, onda se mora referencirati njegovim punim imenom Math.Adding ili pseudonim mora biti potavljen pomoću direktive aliasa.

U Elixir-u nema potrebe da definišemo modul Math, da bismo definisali modul Math.Adding, pošto jezik prevodi sva imena modula u atome. Možemo definisati i proizvoljno ugnježdene module bez defin isanja bilo kog modula u lancu. Na primer, možemo definisati modul Math.Adding.Sum, iako nismo prethodno definisali module Math i Math.Adding.

Direktive

Kako bi se olakšala ponovna upotreba koda, Elixir daje tri direktive - **alias, require i import**. On takođe obezbeđuje i makro **use**.

```
9   #Alias modula tako da se može povati sa Adding umesto Math.Adding
10  alias Math.Adding, as: Adding
11
12  #Obezbeđuje da je modul kompajliran i dostupan (obično za makroe)
13  require Math
14
15  #Uvozi funkcije iz modula Math tako da se mogu pozivati bez prefiksa Math.
16  import Math
17
18  #Uključuje prilagođen kod definisan u Math kao proširenje
19  use Math
```

Slika 1.36:

Aliasi

Direktiva aliasa nam sluzi da podesimo pseudonime za bilo koje ime modula. Aliasi uvek moraju počinjati velikim slovom. Validni su samo unutar leksičkog opsega u kome su pozvani.

Require

Elixir obezbeđuje makroe kao mehanizam za meta-programiranje (pisanje koda koji generiše kod). Makroi su delovi koda koji se izvršavaju i proširuju tokom kompilacije. To znači da bismo koristili makro, moramo garantovati da su njegovi moduli i implementacija dostupni tokom kompilacije. Ovo se čini pomoću require direktive. Uopšteno, moduli nisu potrebni pre upotrebe, osim ako želimo da koristimo makroe, koji su dostupni u njemu. Require direktiva je takođe leksički određena.

Import

Direktivu import koristimo da bismo lakše pristupali funkcijama i makroima iz drugih modula bez upotrebe potpuno kvalifikovanog imena. Import direktiva je takođe leksički određena.

Use

Iako nije direktiva, use je makro koji je usko povezan sa zahtevom koji nam omogućava da koristimo modul u trenutnom kontekstu. Makro use se često koristi od strane progaramera za unos spoljne funkcionalnosti u trenutni leksički opseg, često modula.

Glava 2

Razrada

2.1 Elixir

Elixir je dinamičan, funkcionalni programski jezik koji se pokreće na Erlang vituelnoj mašini, pa samim tim i deli pogodna svojstva kao što su konkurentnost i tolerisanje grešaka, koje dolaze sa ovim okruženjem.[1]

Glava 3

Zaključak

Literatura

[1] Dave Thomas. *Programming Elixir 1.3. The Pragmatic Bookshelf*. 2016.

Biografija autora

Vuk Stefanović Karadžić (*Tršić, 26. oktobar/6. novembar 1787. — Beč, 7. februar 1864.*) bio je srpski filolog, reformator srpskog jezika, sakupljač narodnih umotvorina i pisac prvog rečnika srpskog jezika. Vuk je najznačajnija ličnost srpske književnosti prve polovine XIX veka. Stekao je i nekoliko počasnih mastera. Učestvovao je u Prvom srpskom ustanku kao pisar i činovnik u Negotinskoj krajini, a nakon sloma ustanka preselio se u Beč, 1813. godine. Tu je upoznao Jerneja Kopitara, cenzora slovenskih knjiga, na čiji je podsticaj krenuo u prikupljanje srpskih narodnih pesama, reformu ćirilice i borbu za uvođenje narodnog jezika u srpsku književnost. Vukovim reformama u srpski jezik je uveden fonetski pravopis, a srpski jezik je potisnuo slavenosrpski jezik koji je u to vreme bio jezik obrazovanih ljudi. Tako se kao najvažnije godine Vukove reforme ističu 1818., 1836., 1839., 1847. i 1852.