Programski jezik Elixir

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

Božidar Antić, Marija Novaković,
Srđan Lazarević, Radmila Ninković
bozidar_antic@matf.bg.ac.rs, mi14170@alas.matf.bg.ac.rs,
mi14126@alas.matf.bg.ac.rs, mi14202@alas.matf.bg.ac.rs

29. april 2019

Sažetak

U ovom radu je obrađen funkcionalni programski jezik Elixir. Ukratko je predstavljena istorija jezika, Erlang virtualna mašina i osnove jezika. Objašnjeni su ugrađeni tipovi, anonimne i imenovane funkcije, konkurentnost i moduli. Opisana je instalacija i navedeni su primeri za lakše upoznavanje sa jezikom, kao i poznata okruženja koja se koriste u radu sa Elixir programskim jezikom.

Sadržaj

1 Uvod 3

2 Istorijat 3

3 BEAM - Erlang virtualna mašina 3

4 Osobine jezika 4
- 4.1 Poklapanje obrazaca . . . 4
- 4.2 Imutabilnost . . . 4
- 4.3 Ugrađeni tipovi . . . 5
  - 4.3.1 Atomi, celi i brojevi u pokretnom zarezu i opsezi . . . 5
  - 4.3.2 Procesi, portovi, reference i niske bitova . . . 5
  - 4.3.3 Torke . . . 6
  - 4.3.4 Liste . . . 6
  - 4.3.5 Mape . . . 6
- 4.4 Anonimne funkcije . . . 7
- 4.5 Moduli i imenovane funkcije . . . 7
  - 4.5.1 Pozivi funkcija i poklapanje obrazaca . . . 7
  - 4.5.2 Čuvari (eng. *Guards*) . . . 8
  - 4.5.3 Podrazumevani parametri . . . 8
  - 4.5.4 Operator prosleđivanja |> (eng. *pipe*) . . . 8
  - 4.5.5 Direktive u okviru modula . . . 8
  - 4.5.6 Atributi modula . . . 9
- 4.6 Konkurentnost . . . 9

5 Instalacija **9**
5.1 Linux 9
5.2 Windows 10

6 Primeri **10**

7 Okruženja **11**

8 Zaključak **12**

Literatura **12**

A Dodatak **13**

1 Uvod

Elixir je dinamičan, funkcionalni programski jezik koji se izvršava na Erlang vituelnoj mašini pa samim tim i deli pogodna svojstva kao što su konkurentnost i tolerisanje grešaka, koje dolaze sa ovim okruženjem.[11] Iz tačke gledišta olakšavanja svakodnevnog razvoja softvera, 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 kôd u vreme izvršavanja, polimorfizam, 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.

2 Istorijat

Tokom 2010. godine José Valim, u to vreme zaposlen na poziciji programera u kompaniji Plataformatec, radio je na poboljšanju performansi Ruby on Rails okruženja 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 Erlang-ovoj virtuelnoj mašini podstaklo da započne pisanje Elixir-a. 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: 2015. godine je zabeleženo upravljanje nad 2 miliona WebSocket konekcija [4], dok je 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].

3 BEAM - Erlang virtualna mašina

Prvobitno BEAM je bila skraćenica za Bogdan's Erlang Abstract Machine, po Bogumilu „Bogdan"-u Hausmanu koji je napisao originalnu verziju virtualne mašine, ali ime se takođe može tumačiti kao Björn's Erlang Abstract Machine, po Björn Gustavsson, koji piše i održava trenutnu verziju.

Kako je Elixir zapravo nastao proširivanjem funkcionalnosti BEAM-a (Erlangove virtualne mašine) sa ciljem da se zadrži kompatibilnost sa Erlang ekosistemom. Važno je opisati osnovne koncepte funkcionisanja Erlang virtualne mašine na kojoj se kompajlirani Elixir kôd izvršava.

BEAM predstavlja jezgro Erlang Open Telecom Platform (Erlang/OTP), platforme koja se sastoji od Erlang Run-Time System(ERTS) i kolekcije unapred kompajliranih biblioteka i alata pisanih u Erlangu, sto omogućava Elixir-u direktno korišćenje funkcionalnosti koje pružaju Erlang moduli.

ERTS predstavlja modul zadužen za kompilaciju Erlang i Elixir izvornog kôda u binarni kôd *(eng. bytecode)* koji se onda izvršava u okviru BEAM-a.

U okviru BEAM-a sav kôd se izvršava u sitnim konkurentnim procesima, pri čemu procesi nisu procesi operativnog sistema vec Erlang procesi. To je moguće upravo zbog izvršavanja u okviru BEAM virtualne

mašine. Svaki od procesa je memorijski nezavistan i sva međuprocesna komunikacija se odvija razmenjivanjem poruka, čime se obezbeđuje koncept konkurentnosti slanjem poruka *(eng. message passing concurrency)*.[11]

Prevođenjem Elixir izvornog fajla generiše se *.beam* fajl koji sadrži binarni kôd *(eng. bytecode)* koji se može izvršavati u okviru BEAM-a.

4 Osobine jezika

U ovom poglavlju će biti opisane osobine Elixir-a, 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 Elixir-u. 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 preklapanja obrazaca itd.[7]

4.1 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 Elixir-u se naziva operator *uparivanja* (eng. *matching*). On se uspešno izvršava ako pronađe način da izjednači levu stanu (svoj prvi operand) sa desnom (drugi operand).

Primer 4.1 *U narednom delu kôda a je promenljiva, koju Elixir uspeva da izjednači sa vrednoću 1, tako što biva postavljena na tu vrednost. U drugoj naredbi izvršavanje operacije ponovo uspeva jer je vrednost leve strane 1 dok promenljiva sa desne strane već ima vrednost 1 zbog prethodne operacije. Sledeća naredba ne uspeva jer Elixir pokušava da izjednači levu stranu, koja je broj 2 i može imati samo tu vrednosti, sa promenjivom a koja ima vrednost 1. Ključna stvar za razumeti je da Elixir ne pokušava da izjednači levu i desnu stranu, već levu sa desnom.*

```
iex> a = 1
#out: 1
iex> 1 = a
#out: 2
iex> 2 = a
#out:(MatchError) no match of right hand side value:
     1
```

Listing 1: Primer jednostavnog poklapanja obrazaca

4.2 Imutabilnost

Imajući na umu funkcionalnu paradigmu Elixir-a, imutabilnost podataka je osobina koja je neizostavna. Naime, jednom kreirani podaci ne mogu biti promenjeni. Bez ulaženja u dublje analize rasprava o tome da li je mutabilnost podataka dobra ili loša, činjenica je da postojanje stanja

i funkcije koje to stanje mogu menjati u izrazima sa bočnim efektima. Otežava razumevanje i još bitnije, utvrđivanje korektnosti funkcionisanja sistema. Pogotovo u slučaju višenitnih, konkurentnih ili distribuiranih programa kakvi se najčešće pišu korišćenjem Elixir programskog jezika.

4.3 Ugrađeni tipovi

Elixir implementira desetine tipova. Od njih je važno istaći ugrađene - primitivne tipove, preko kojih su realizovane implementacije ostalih:

- Atomi (*Atom*)
- Celi brojevi (*Integer*)
- Brojevi u pokretnom zarezu (*Float*)
- Procesi (*Process*)
- Portovi (*Port*)
- Uređene torke (*Tuple*)
- Liste (*List*)
- Mape (*Map*)
- Funkcije (*Function*)
- Niske bitova
- Reference

U zagradama nakon tipa, osim u poslednja dva koji nemaju odgovarajuće module, navedena su imena modula koji sadrže funkcije koje se koriste za operacije nad tim tipom. Imena ovih modula ne treba mešati sa primitivnim tipovima navedenim gore, iako oni sami predstavljaju tip. Oni se mogu zamisliti kao neka vrsta omotača oko primitivnog tipa koji obezbeđuje bogatije funkcionalnosti nad njime.

Može biti čudno što se na ovoj listi nisu našle niske ili strukture, ali one su deo složenih tipova podržanih od strane Elixir-a. Takođe, postoji debata o tome da li su regularni izrazi i opsezi (*Ranges*) tipovi za sebe. U nekoj literaturi se posmatraju ovako, iako su tehnički strukture. [11]

4.3.1 Atomi, celi i brojevi u pokretnom zarezu i opsezi

Atomi su konstante koje predstavljaju ime. Počinju dvotačkom i njihovo ime im je i vrednost, što znači da će dva atoma sa istim imenom pri poređenju uvek biti isti bez obzira na to da li su kreirani od strane različitih aplikacija ili procesa.

Celi brojevi su slični kao i u većini drugih jezika i mogu se obeležavati korišćenjem dekadne (1234), heksadekadne (0xcafe), oktalne (0o1234) i binarne (0b1010) notacije. Karakter _ se može koristiti za odvajanje blokova cifara. Bitna stvar je da ne postoji fiksna veličina za čuvanje celih brojeva u memoriji, već interna reprezentacija raste kako bi obezbedila da broj bude smešten u celosti.

Brojevi u pokretnom zarezu se u memoriji zapisuju po standardu IEEE 754 [11][2], a za zapisivanje konstanti ovog tipa koristi se tačka između najmanje dve cifre. Takođe je moguće koristiti i notaciju koja obuhvata navođenje eksponenta.

4.3.2 Procesi, portovi, reference i niske bitova

Procesi predstavljaju pogodnost za rad sa nitima.

Portovi su reference na spoljne resurse koji omogućavaju interakciju sa spoljnim svetom.

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

4.3.3 Torke

Torke su zamišljene kao strukture fiksne dužine koje bi trebalo da sadrže svega nekoliko elemenata. Osnovna stvar koja ih razlikuje od *listi* je u semantici njihove upotrebe. *Liste* se koriste kada se manipuliše kolekcijom, dok se operacija čitanja elementa iz *torke* izvršava u konstantnom vremenu, pa se uglavnom koristi kako bi se u nju smestile povratne vrednosti funkcije. Može da sadrži elemente različitog tipa i ove vrednosti su u memoriji zapisane jedna za drugom.

4.3.4 Liste

Lista je kolekcijski tip koji je imlementiran kao povezana lista. Ova osobina čini da je pristup proizvoljnom elementu složenosti $O(n)$. Lista može biti prazna ili sastavljena od glave i tela, gde je glava element liste, a telo je lista. Primećuje se rekurzivna definicija liste koja je zajedno sa operacijom izdvajanja glave od tela (koja je pritom efikasna znajući strukturu implementacije) osnov za mnoge programerske prakse u Elixir-u. Pošto je lista imutabilna, kao i svi podaci u Elixir-u, čitalac može pomisliti da će pri prethodno navedenoj operaciji biti napravljena nova lista koja će predstavljati rep prethodne. Ovo bi bilo memorijski, pa i vremenski vrlo neefikasno pa je u Elixir-u implementirano tako što će umesto nove liste biti vraćen pokazivač na rep prethodne.

```
iex> list = [1,2 | [3]]
#out: [1,2,3]
iex> list.last([1, 2, 3, 4])
#out: 4
```

Listing 2: Primer dodavanja na kraj liste i ispisa poslednjeg elementa

4.3.5 Mape

Mapa je kolekcija koja sadrži parove ključeva i vrednosti. Glavna razlika između liste parova ključeva i vrednosti, i mape leži u tome da je mapa asocijativna struktura podataka i samim tim ne dozvoljava duplirane vrednosti ključa. Mapa je vrlo efikasna, pogotovo sa rastom količine podataka. Ukoliko je potrebno da u kolekciji podaci ostanu u redosledu u kome su navedeni pri inicijalizaciji, onda je lista parova bolji izbor. One se najčešće koriste kako bi se funkcijama prosledile opcije. U ostalim slučajevima, mapa je uglavnom bolji izbor.

Primer 4.2 *Primer inicijalizacije liste parova ključeva i vrednosti, i mape sa istim vrednostima. Za listu je korišćena skraćenica obezbeđena u Elixir-u zbog čestog koriščenja ovog kontrukta. Parovi u listi su tipa torke, ključevi su atomi, dok su vrednosti niske.*

```
list = [key1: "value1", key2: "value2", key3: "
    value3"]
# interno: [ {:key1,"value1"}, {:key2,"value2"}, {:
    key3,"value3"} ]
map = %{:key1 => "value1",:key2 => "value2",:key3 =>
     "value3"}
```

Listing 3: Primer liste parova ključeva i vrednosti, i mape

4.4 Anonimne funkcije

Anonimne funkcije su u Elixir-u kao i u drugim funkcionalnim programskim jezicima građani prvog reda. Iako eksplicitno nisu imenovane, na funkcije ove vrste se možemo referisati tako što ih sačuvamo u promenljivoj. Takođe, zanimljivo je da anonimne funkcije mogu imati više tela, odnosno različite implementacije u zavisnosti od broja ili vrednosti parametara. Ova funkcionalnost, koja poznavaoce objektno orijentisanog programiranja može podsetiti na koncept preopterećivanja (eng. *overloading*) metoda, je ostvarena pomoću poklapanja obrazaca.

Primer 4.3 *Naredna anonimna funkcija pokušava da ispiše prvu liniju fajla čije je ime prosleđeno kao argument, a ukoliko fajl ne postoji ispisuje poruku o grešci.*

```
handle_open = fn
    {:ok, file} -> "Read data: #{IO.read(file, :line
    )}"
    {_, error}  -> "Error: #{:file.format_error(
    error)}"
end
```

Listing 4: Primer anonimne funkcije

4.5 Moduli i imenovane funkcije

Kada program sadrži previše linija kôda, poželjno je strukturirati ga. U Elixir-u kôd se može izdvojiti u imenovane funkcije koje se dalje organizuju u module, čak se moraju pisati u okviru njih. Jednostavan primer modula Times koji sadrži jednu funkciju double prikazan je ispod.

```
defmodule Times do
    def double(n) do
        n * 2
    end
end
```

Listing 5: Množenje broja sa 2

Jedan način grupisanja izraza i prosleđivanje istih drugom kôdu je *do..end* blok. Koristi se u modulima, imenovanim funkcijama, strukturama kontrole (eng. *control structures*) i na bilo kom drugom mestu gde kôd treba da se koristi kao entitet. Može se proslediti i više linija grupisanjem zagradama:

```
def greet(greeting, name), do: (
    IO.puts greeting
    IO.puts "How're you doing, #{name}?"
)
```

Listing 6: Ispisivanje poruke pozdrava

4.5.1 Pozivi funkcija i poklapanje obrazaca

Za razliku od anonimnih funkcija u ovom slučaju moramo pisati funkciju više puta, svaki put sa listom parametara i telom funkcije. Kada se funkcija pozove, Elixir pokušava da poklopi argumente sa listom parametara prve definicije, ukoliko ne može da ih poklopi nastavlja sa sledećom

definicijom iste funkcije itd. dok ne nađe pravog kandidata. Važno je obratiti pažnju na redosled pisanja funkcija jer Elixir uvek pokušava od vrha ka dole i izvršava se prva koja se poklopila.

4.5.2 Čuvari (eng. *Guards*)

Poklapanje obrazaca omogućava Elixir-u da odluči koju funkciju će pozvati na osnovu argumenata koji su prosleđeni. Međutim ukoliko treba razlikovati na osnovu tipova ili na osnovu nekih testova koji uključuju vrednosti koristimo čuvare. To su zapravo predikati koji su pridruženi definiciji funkcije koristeći ključnu reč *when* jednom ili više puta.

4.5.3 Podrazumevani parametri

Kada se definiše imenovana funkcija može se dati podrazumevana vrednost bilo kom parametru korišćenjem sintakse param \\vrednost. Pri pozivu funkcije koja ima podrazumevane vrednosti, poredi se broj argumenata koji se prosleđuju sa brojem obaveznih vrednosti. Ukoliko je broj argumenata manji nema poklapanja. Ukoliko je njihov broj jednak, obavezne vrednosti uzimaju vrednosti argumenata, a ostali parametri uzimaju podrazumevane vrednosti. U slučaju da je broj prosleđenih argumenata veći od obaveznih, Elixir može da pregazi podrazumevane vrednosti nekih ili svih parametara tako što ide sa leva na desno.

```
defmodule Example do
    def func(p1, p2 \\ 2, p3 \\ 3, p4) do
        IO.inspect [p1, p2, p3, p4]
    end
end
Example.func("a", "b") # => ["a",2,3,"b"]
Example.func("a", "b", "c", "d") # => ["a","b","c","
    d"]
```

Listing 7: Poziv funkcije sa podrazumevanim parametrima

4.5.4 Operator prosleđivanja |> (eng. *pipe*)

Operator |> uzima rezultat izraza sa leve strane i ubacuje ga kao prvi parametar poziva funkcije sa desne strane. U suštini val |> f(a, b) je isto što i f(val, a, b).

4.5.5 Direktive u okviru modula

- Import direktiva - dovlači funkcije i makroe nekog modula u trenutni opseg, eliminiše se potreba za ponavljanjem imena modula. *Only:* ili *except:* su opcioni parametri praćeni listom parova *naziv:broj* parametara. Pogodni su za kontrolu koje funkcije ili makroi su uključeni (eng. *import*).

  ```
  import Module [, only: | except: ]
  ```

- Alias direktiva - kreira alias za modul i samim tim skraćuje pisanje. Na primer:

  ```
  alias My.Other.Module.Parser, as: Parser
  ```

- Require direktiva - obezbeđuje da su definicije makroa dostupne prilikom prevođenja.

4.5.6 Atributi modula

Atributom modula se naziva svaka stavka metapodatka i identifikovan je imenom. U okviru modula može se pristupiti atributu stavljajući prefiks '@' ispred imena. Dodeljivanje vrednosti atributu vrši se sa @ime vrednost i ono se ne može obavljati u okviru funkcije. Ovi atributi nisu promenljive u konvencionalnom smislu i uglavnom se upotrebljavaju kao što Java ili Ruby programeri upotrebljavaju konstante.

4.6 Konkurentnost

Jedna od glavnih karakteristika Elixir-a je ideja o pakovanju kôda u male celine koje se mogu pokretati nezavisno i istovremeno. Elixir za konkurentnost koristi model *actor*. *Actor* je nezavistan proces koji ništa ne deli ni sa jednim drugim procesom. Mogu se napraviti novi, poslati im poruke i primiti poruke nazad. Procesi se mogu izvoditi na svim jezgrima procesora, imaju malo dodatnih troškova. Lako je napraviti i nekoliko stotina hiljada procesa čak i na slabijim računarima.

Procesi podrazumevano nisu povezani, pa samim tim nisu svesni stanja kroz koje prolaze drugi procesi. Pomoću funkcije spawn_link pravi se novi proces koji je povezan sa roditeljskim i kroz mehanizme povezivanja (eng. *linking*) i nadgledanja (eng. *monitoring*) mogu komunicirati.

Ukoliko želimo da obezbedimo distribuirano izvršavanje svojih programa, Elixir obezbeđuje koncept čvorova izvršavanja. Čvorovi nisu svesni postojanja drugih dok se eksplicitno ne povežu. Ukoliko su čvorovi povezani, izvršavanje i npr. obrada rezultata jednog procesa se mogu odvijati na različitim, potencijalno kilometrima udaljenim računarima. [11]

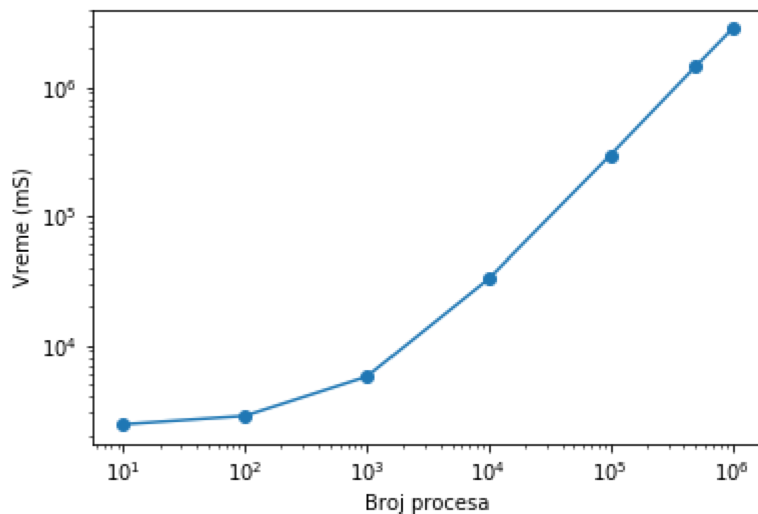
Na slici 1 prikazana je zavisnost vremena potrebnog da se procesi naprave u odnosu na njihov broj. Grafik je iscrtan na logaritamskoj skali pa na prvi pogled možda nije očigledno da rast prati linearni trend. Informacije o programu koji je pokretan kao i mašini na kojoj je izvršavan nalaze se u Dodatku A.

5 Instalacija

Pre instalacije Elixir-a neophodno je instalirati Erlang na odgovarajućem operativnom sistemu. Mnogi različiti upravljači paketima (apt, dnf, pkg, pacman, MacPorts, Homebrew, itd.) uključuju Erlang. On je sve više deo standardne instalacije na mnogim sistemima, uključujući i Ubuntu, uglavnom zahvaljujući širenju CouchDB-a. Nakon instalacije Erlang-a može se instalirati Elixir.

5.1 Linux

Ako koristite operativni sistem Linux, Elixir se može instalirati komandom *sudo apt-get install elixir*[7]. Nakon instalacije, Elixir programi se mogu kompajlirati i pokretati u interpreteru. Kôd iz datoteke se može kompajlirati komandom *elixir* iza koje se navodi ime fajla sa ekstenzijom *.exs*.



Slika 1: Grafik potrebnog vremena za pravljenje procesa

5.2 Windows

Ako koristite operativni sistem Windows, instaliranje Elixir-a je jednostavno. Sa oficijalnog sajta Elixir-a [7] se preuzme binarna datoteka i prati se uputstvo instalera do završetka instalacije.

6 Primeri

Prvi primer za upoznavanje svakog programskog jezika jeste „Pozdrav svete!". Definisan je modul *ModuleName* i u okviru njega funkcija *hello* koja ispisuje poruku.

```
defmodule ModuleName do
  def hello do
    IO.puts "Hello World!"
  end
end
```

Listing 8: Ispis poruke "Hello World!"

Modul *Guard* opisuje korišćenje ključne reči *when* koja proverava zadovoljenost uslova koji sledi. Na standardni izlaz se ispisuje tip prosleđenog argumenta.

```
defmodule Guard do
    def what_is(x) when is_number(x) do
        IO.puts "#{x} is a number"
    end
    def what_is(x) when is_list(x) do
        IO.puts "#{inspect(x)} is a list"
    end
end
```

Listing 9: Provera da li je argument broj ili lista

Modul *Fib* prikazuje poklapanja obrazaca funkcija pri čemu se mora voditi računa o redosledu definicija funkcija, jer bi u protivnom moglo doći do beskonačne rekurzije.

```
defmodule Fib do
  def fib(0) do 0 end
  def fib(1) do 1 end
  def fib(n) do fib(n-1) + fib(n-2) end
end
```

Listing 10: Fibonačijev niz

7 Okruženja

Elixir je dizajniran da bude proširiv, omogućuje programerima da prirodno prošire jezik na određene domene, kako bi povećali svoju produktivnost. Danas se najviše koristi za veb programiranje, ugradnih uređaja (eng. *embedded devices*), dok svoju upotrebu pronalazi i u projektima vezanim za robotiku. Sa konstantnim razvitkom jezika, javljaju se i okruženja koja pronalaze primenu u oblasti mašinskog učenja [8]. Neka Elixir okruženja otvorenog koda (eng. *open source*) su:

- Phoenix Framework[4] - Pruža jednostavnost u pisanju veb aplikacija kombinovanjem poznatih i poverljivih tehnologija sa dodatnim funkcionalnim idejama. Ovo okruženje koristi MVC (eng. Model View Controller) obrazac [4] i Cowboy server [9] za obradu zahteva.
- Nerves [3] - Definiše novi način za izgradnju ugrađenih sistema koristeći Elixir. Posebno dizajniran za ugrađene sisteme i sastoji se iz tri komponente:
  - Platforma - napravljena po zahtevu, minimalni Linux sistem izveden iz Buildroot-a koji se pokreće direktno na BEAM
  - Okvir (eng. *Framework*) - spremna biblioteka modula Elixir-a za brzo pokretanje
  - Alati (eng. *Tooling*) - moćne alatke komandne linije za upravljanje gradnjom, ažuriranje firmware-a, konfigurisanje uređaja i još mnogo toga
- Hedwig - Dizajniran za dva slučaja upotrebe:
  - jedinstvene, samostalne Erlang/OTP aplikacije
  - uključen kao zavisnost od drugih Erlang/OTP aplikacija

  Hedwig se isporučuje sa adapterom za konzolu kako bi omogućio brzo pokretanje. Odličan je za testiranje kako će bot odgovoriti na poruke koje dobije.
- Plug se koristi kao nivo apstrakcije odnosno adapter za konekciju na različite veb servere u BEAM.
- Trot je mikro-okvir baziran na Plug-u i Cowboy-u. Cilj Trot-a je da olakša upotrebu uobičajnih obrazaca u Plug-u, posebno prilikom pisanja API-ja.
- Anubis omogućuje kreiranje aplikacije komandne linije uz pomoć više komandi (kao što je git), bez potrebe da definiše više miks zadataka.
- Još neki okviri u upotrebi su Placid, Kitto, Maru, Sugar, Urna i Flowex

Tabela 1: Elixir okruženja otvorenog koda.

	Web	API	CMD Line Application	Embedded Software	Bot Creation
Phoenix	x	x			
Nerves				x	
Sugar	x				
Hedwig					x
Plug	x				
Trot	x				
Placid		x			
Anubis			x		

8 Zaključak

Zaključak ovog rada možda se najbolje može opisati citiranjem samog autora José-a Valim-a, koji o Elixir-u kaže sledeće:

„Elixir predstavlja pragmatičan pristup funkcionalnom programiranju. Vrednuje svoje funkcionalne korene i fokusira se na produktivnost programera. Konkurentnost je okosnica softvera razvijenog pomocu Elixir-a. Kao što je sakupljanje otpada oslobodilo programere iz okova upravljanja memorijom, tako je Elixir tu da vas oslobodi od zastarelih konkurentnih mehanizama i donosi vam radost prilikom pisanja konkurentnog kôda." [11]

Literatura

[1] Discord organization. on-line at: https://discordapp.com/open-source.

[2] IEEE organization. on-line at: https://www.ieee.org/.

[3] Nerves project documentation. on-line at: https://nerves-project.org/.

[4] Phoenix framework documentation. on-line at: https://phoenixframework.org/.

[5] Pinterest. on-line at: https://github.com/pinterest.

[6] Plataformatec. on-line at: https://plataformatec.com/.

[7] José Valim. Elixir. on-line at: https://elixir-lang.org/.

[8] José Valim, Anshuman Chhabra. Tensorflex: Tensorflow bindings for the Elixir programming language. In *32nd Conference on Neural Information Processing Systems (NIPS)*, 2018.

[9] McCord Chris, Tate Bruce, Valim José. *Programming Phoenix*. The Pragmatic Bookshelf, Raleigh, North Carolina, 2016.

[10] Simon St. Laurent, J. David Eisenberg. *Introducing Elixir*. O'Reilly Media, 2014.

[11] Dave Thomas. *Programming Elixir 1.3*. The Pragmatic Bookshelf, Raleigh, North Carolina, 2016.

A Dodatak

Program koji je korišćen za testiranje brzine pravljenja procesa nalazi se u nastavku. Pokretan je u terminalu pomoću:

```
elixir --erl "+P 1000000" -r chain.exs -e " Chain.run(N)"
```

gde je umesto N upisivan željeni broj procesa koje treba napraviti. Računar na kome je pokretan program je marke HP Omen 15, Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 16Gb RAM, sa Ubuntu 18.04 operativnim sistemom.

```
defmodule Chain do
    def counter(next_pid) do
          receive do
                  n -> send next_pid, n + 1
                  end
        end
    def create_processes(n) do
          last = Enum.reduce 1..n, self,
                     fn (_,send_to) -> spawn(Chain,
    :counter, [send_to])
                                    end
          send last, 0
          receive do
                  final_answer when is_integer(
    final_answer) -> "Result is #{inspect(
    final_answer)}"
                  end
        end
    def run(n) do
          IO.puts inspect :timer.tc(Chain, :
    create_processes, [n])
        end
end
```

Listing 11: chain.exs