

UNIVERZITET U BEOGRADU  
MATEMATIČKI FAKULTET

Nemanja Subotić

PROGRAMSKI JEZICI ELM I ELIXIR U  
RAZVOJU STUDENTSKOG VEB PORTALA

master rad

Beograd, 2020.

**Mentor:**

dr Milena VUJOŠEVIĆ JANIČIĆ, docent  
Univerzitet u Beogradu, Matematički fakultet

**Članovi komisije:**

dr Filip MARIĆ, vanredni profesor  
Univerzitet u Beogradu, Matematički fakultet

dr Ivan ČUKIĆ, docent  
Univerzitet u Beogradu, Matematički fakultet

**Datum odbrane:** \_\_\_\_\_

**Naslov master rada:** Programski jezici Elm i Elixir u razvoju studentskog veb portala

**Rezime:** Apstrakt rada

**Ključne reči:** elm, elixir, ...

# Sadržaj

<b>1</b>	<b>Uvod</b>	<b>1</b>
<b>2</b>	<b>Programski jezik i okruženje Elm</b>	<b>2</b>
2.1	Uputstvo za instalaciju . . . . .	3
2.2	Osnovne odlike . . . . .	3
2.3	Elm kao platforma . . . . .	4
2.4	Elm kao jezik . . . . .	5
2.5	Arhitektura Elm . . . . .	20
<b>3</b>	<b>Elixir</b>	<b>24</b>
3.1	Erlang . . . . .	25
3.2	Osnovne karakteristike jezika Elixir . . . . .	27
3.3	Uputstvo za instalaciju . . . . .	27
3.4	Osnovni tipovi podataka . . . . .	27
3.5	Osnovni operatori . . . . .	34
3.6	Poklapanje obrazaca . . . . .	36
3.7	Funkcije . . . . .	37
3.8	Makroi . . . . .	43
3.9	Kontrola toka . . . . .	44
3.10	Polimorfizam preko protokola . . . . .	47
<b>4</b>	<b>Implementacija MSNR portala</b>	<b>48</b>
<b>5</b>	<b>Zaključak</b>	<b>49</b>
	<b>Bibliografija</b>	<b>50</b>

# Glava 1

## Uvod

Funkcionalno programiranje kao programska paradigma nastaje 1959.godine sa pojavom LISP-a, prvog funkcionalnog programskog jezika... Elm... Phoenix i Elixir... MSNR Poral...

## Glava 2

# Programski jezik i okruženje Elm

Evan Zaplicki (Evan Czaplicki) je 2012. godine objavio svoju tezu „Elm: Konkurentno FRP <sup>1</sup> za funkcionalne GUI-je <sup>2</sup>” (eng. „*Elm: Concurrent FRP for Functional GUIs*”) [1] i, s ciljem da GUI programiranje učini prijatnijim, dizajnirao novi programski jezik — Elm. Na slici 2.1 prikazan je logo jezika. Elm je statički tipiziran, čisto funkcionalni programski jezik koji se kompilira, tačnije transpilira u JavaScript i namenjen je isključivo za kreiranje korisničkog interfjesa veb aplikacija. Takođe,



Slika 2.1: Logo programskog jezika Elm

Elm nije samo programski jezik već i platforma za razvoj aplikacija. Zbog svoje funkcionalne prirode i prisustva kompilatora, Elm spada među najstabilnija i najpouzdanija razvojna okruženja, a za Elm aplikacije važi da, u praksi, ne izbacuju neplanirane greške tokom izvršavanja (eng. *No Runtime Exceptions*).

---

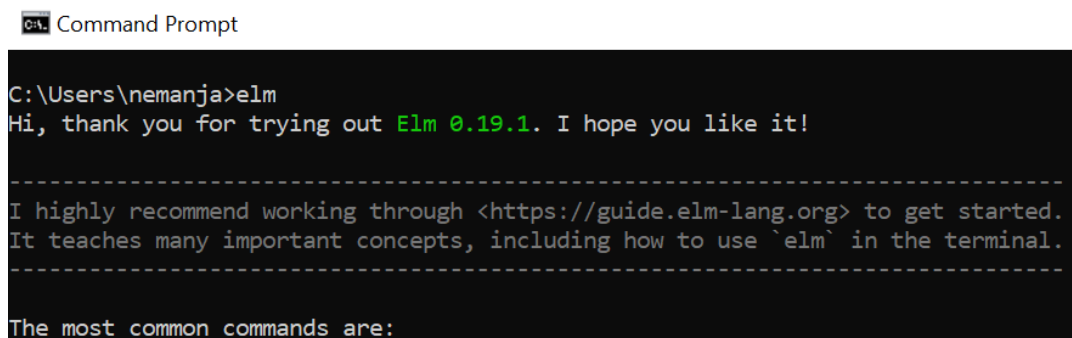
<sup>1</sup>FRP je skraćenica za funkcionalno reaktivno programiranje

<sup>2</sup>GUI je skraćenica za grafički korisnički interfjes

## 2.1 Uputstvo za instalaciju

Pored želje da frontend programiranje učini prijatnijim, kreator jezika nastoji da ono bude i pristupačnije. Stoga, da bi se počelo sa korišćenjem Elma instalacija nije potrebna, dovoljeno je otići na zvaničnu veb stranicu i pokrenuti dostupan interaktivni kompilator [15], gde se može naći dosta primera, kao i vodič kroz Elm.

Za zahtevnije projekte neophodno je izvršiti instalaciju, koja je vrlo jednostavna. Potrebno je pratiti instrukcije sa zvanične stranice [6]. Takođe, moguća je instalacija pomoću **npm**<sup>3</sup> alata [11]. Na slici 2.2 prikazana je provera uspešne instalacije, koja se može izvršiti pokretanjem komande **elm** u komandnoj liniji, gde će se prikazati poruka dobrodošlice i spisak mogućih komandi o kojima će biti reč u sledećim poglavljima.



```
C:\> Command Prompt
C:\Users\nemanja>elm
Hi, thank you for trying out Elm 0.19.1. I hope you like it!

-----
I highly recommend working through <https://guide.elm-lang.org> to get started.
It teaches many important concepts, including how to use `elm` in the terminal.
-----

The most common commands are:
```

Slika 2.2: Provera uspešne instalacije Elma

## 2.2 Osnovne odlike

Pored *No Runtime Exceptions*, jedna od glavnih odlika ovog jezika jeste kompilator, koji je izuzetno ugodan za rad. Mnogi programeri smatraju da Elm kompilator proizvodi najbolje poruke o greškama. Za razliku od drugih, Elm kompilator objašnjava zašto je došlo do greške i daje predloge za njihovo rešavanje, a takođe nema kaskadnih poruka. Kreator se vodio razmišljanjem da kompilator treba da bude asistent, ne samo alat.

Elm koristi svoju verziju *virtualnog DOM*<sup>4</sup>-a, koncepta koji se koristi u mnogim frontend okruženima. Ideja je da se u memoriji čuva „virtualna” reprezentacija

---

<sup>3</sup>npm — *Node Package Manager* predstavlja alat za upravljanje paketima u JavaScript programskom jeziku

<sup>4</sup>DOM — Obejktni model dokumenta (eng. *Document Object Model*)[12]

korisničkog interfejsa na osnovu koje se ažurira „stvarni” DOM. Još jedna bitna karakteristika Elma je nepromenljivost podataka, što znači da se jednom definisani podaci ne mogu više menjati. Direktna posledica nepromenljivosti podataka je veoma brzo renderovanje HTML-a, jer se poređenja u virtuelnom DOM-u mogu vršiti po referenci. Verzija Elm 0.17 imala je najbrže renderovanje u poređenju sa tadašnjim aktuelnim verzijama popularnih okruženja[3].

Elm se može integrisati i u postojeće JavaScript projekte za implementaciju pojedinačnih komponenti. Takođe, moguća je i komunikacija između Elma i JavaScripta.

## 2.3 Elm kao platforma

Elm sa sobom donosi niz alata (tabela 2.1) i Elm okruženje (eng. *Elm Runtime*), koji su neophodni za razvoj i izvršavanje aplikacija. Elm kôd se nalazi u datotekama sa *.elm* ekstenzijom i prilikom kompilacije kreira se jedna izlazna *.js* datoteka. U izlaznoj datoteci se pored prevedenog koda iz ulaznih *.elm* datoteka nalaze i funkcije iz Elm okruženja potrebne za izvršavanje programa.

Alati	Kratat opis
<b>repl</b>	Pokretanje interaktivne sesije (eng. <i>Read-Eval-Print-Loop</i> )
<b>init</b>	Inicijalizacija projekta
<b>reactor</b>	Pokretanje lokalnog servera
<b>make</b>	Upotreba kompilatora
<b>install</b>	Preuzimanje paketa
<b>diff</b>	Prikazivanje razlika između različitih verzija istog paketa
<b>bump</b>	Određivanje broja naredne verzije paketa
<b>publish</b>	Objavljivanje paketa

Tabela 2.1: Elm alati komandne linije

Kao zaseban jezik Elm ima i zaseban sistem za upravljanje paketima. Pokretanjem komande **elm init** kreira se prazan *src* direktorijum i datoteka *elm.json*, u kojoj se pored informacije o tipu projekta (aplikacija ili paket), Elm verzije i liste direktorijuma sa kodom, nalazi i spisak paketa koji se koriste u projektu. Dodavanje novog paketa se vrši pomoću komande **elm install naziv-paketa**. Svi paketi su javno dostupni (<https://package.elm-lang.org/>), nazivi paketa su oblika *autor/ime-paketa*.

Kompilacija se vrši naredbom **elm make <jedna-ili-više-elm-datoteka>**, ukoliko se ne navede izlazna datoteka pomoću argumenta *--output* generisaće se *index.html*



datoteka sa prevedenim JavaScript kodom. Ostali argumenti kao i više informacija o drugim alatima mogu se videti pomoću naredbe `elm naziv-alata --help`

## 2.4 Elm kao jezik

Kao i većina statički tipiziranih funkcionalnih programskih jezika, Elm se zasniva na programskom jeziku ML, a budući da su u programskom jeziku Haskell napisani Elm kompilator i ostali alati, Haskell je ostavio veliki uticaj i na sam jezik. Autor Elma smatra:

„Rekao bih da je Elm ML sa sintaksom poput Haskell-a. Ako poredimo semantiku, Elm je dosta sličniji OCaml-u i SML-u.” [2]

**ML** (eng. *Meta Language*)[7] je statički tipiziran programski jezik opšte namene koji je nastao 1973. godine na Univerzitetu u Edinburgu. Vođa grupe koja je radila na dizajniranju programskog jezika ML bio je Robin Milner, dobitnik Tjuringove nagrade. Nastao je pod uticajem programskog jezika LISP, a razvijan je za implementiranje automatskog dokazivača teorema. Osnovna karakteristika jeste uvođenje automatskog zaključivanja tipova, a odlikuje ga i poklapanje obrazaca, Karijeve funkcije i posedovanje sakupljača otpadaka. ML nije čist funkcionalan jezik i nema ugrađenu podršku za lenjo izračunavanje. U porodicu ML jezika, između ostalih, spadaju i **Standard ML**, **OCaml** i **F#**.

**Haskell**[5] je čist funkcionalni programski, naziv je dobio po matematičaru i logičaru Haskelu Bruks Kariju (eng. *Haskell Brooks Curry*). Haskell je strogo tipiziran, poseduje automatsko zaključivanje tipova i lenjo izračunavanje. Jezik je opšte namene, pruža podršku za paralelno i distirbuirano programiranje. Haskell omogućava manje grešaka i veću pouzdanost kroz kraći i čistiji kôd, koji je lakši za održavanje.

## Komentari

Komentari se u Elmu mogu navoditi na dva načina:

- Korišćenjem `--` za linijske komentare
- Navođenjem teksta između znakova `{- i -}` za komentare u više redova.

```
> 'Z'
'Z' : Char
> "Zdravo!"
"Zdravo!" : String
> True
True : Bool
> 42
42 : number
> 42 / 10
4.2 : Float
> 42 // 10 --celobrojno deljenje
4 : Int
```

Listing 1: Osnovni tipovi podataka prikazani u interpreteru

## Osnovni tipovi podataka

Osnovni tipovi podataka u Elmu su **Char**, **String**, **Bool**, **Int** i **Float**. U listingu 1 prikazani su osnovni tipovi korišćenjem interpretera (`elm repl`). Budući da i Elm poseduje zaključivanje tipova, nakon izračunate vrednosti unetog izraza ispisuje se tip. U konkretnom primeru broj 42 se može posmartati i kao tip **Int** i kao tip **Float**, pa interpreter vraća **number** kao tip, iako **number** nije konkretan tip podataka.

Tip **Char** služi za predstavljanje unikod (eng. *unicode*) karaktera. Karakteri se navode između dva apostorfa ('a', '0', '\t'...), a moguće je koristiti i unikod zapis '\u{0000}' - '\u{10FFFF}'.

Za razliku od Haskell-a, gde je **String** zapravo lista karaktera, u Elmu je poseban tip i predstavlja sekvencu unikod karaktera. Sekvenca se navodi između jednostrukih ili trostrukih navodnika (listing 2).

```
> "\t String u jednom redu: escape navodnici \"Zdravo!\""
"\t String u jednom redu: escape navodnici \"Zdravo!\"" : String
>
> """String u više redova
  sa "navodnicima"! """
"String u više redova\n  sa \"navodnicima\"! " : String
```

Listing 2: Primeri stringova

Tip **Bool** predstavlja logički tip i može imati vrednost **True** ili **False**.

Tip **Int** se koristi za prikazivanje celih brojeva. Siguran opseg vrednosti je od  $-2^{31}$  do  $2^{31} - 1$ , van toga sve zavisi od cilja kompilacije. Ukoliko je cilj kompilacije JavaScript (što je trenutni slučaj), opseg se proširuje na  $-2^{53}$  do  $2^{53} - 1$ . Do

proširivanja opsega ne bi dolazilo ukoliko bi se umesto JavaScript koda generisao *WebAssembly*, tada bi postojalo prekoračenje celih brojeva (eng. *integer overflow*). Vrednosti se mogu navoditi i u heksadecimalnom obliku (`0x2A`, `-0x2b`).

Tip **Float** služi za predstavljanje brojeva u pokretnom zarezu po standardu *IEEE 754*. Vrednosti se mogu navoditi i pomoću eksponencijalnog zapisa, a decimalna tačka se mora nalaziti između dve cifre. Takođe, u skup vrednosti spadaju `NaN` i `Infinity` (listing 3).

```
> 1e3
1000 : Float
> 0/0
NaN : Float
> 1/0
Infinity : Float
```

Listing 3: Prikaz brojeva u pokretnom zarezu

## Osnovni operatori

Kod aritmetičkih operacija, operatori `+`, `-`, `*` se mogu koristiti sa realnim i celim brojevima, dok imamo posebne operatore za deljenje (`/` i `//` koji su i ranije prikazani u listingu 1). Elm ne podržava implicitne konverzije tipova, pa prilikom sabiranja celog broja sa realnim, bez eksplicitne konverzije, kompilator prijavljuje grešku (listing 4). Postoji još i eksponencijalni operator `^`, a za celobrojno deljenje sa ostatkom koriste se funkcije `modBy` i `remainderBy`.

```
> toFloat (9 // 3) + 3.2
6.2 : Float
> 9 // 3 + round 3.2
6 : Int
> 9 // 3 + 3.2 -- TYPE MISMATCH error
```

Listing 4: Upotreba eksplicitne konverzije tipova

Elm pruža logičke operatore `i && i` ili `||` kao i funkcije za negaciju `not` i ekskluzivno ili `xor`. Operator `&&` ima viši prioritet od operatora `||`, oba su levo asocijativna i lenjo izračunljiva. Od operatora poređenja `==`, `/=`, `<`, `>`, `>=` i `<=`, jedino operator različitosti (`/=`) ima drugačiju sintaksu od uobičajene. Pored navedenih, Elm podržava i operator `++` koji se koristi za konkatenciju stringova i listi. Listing 5 prikazuje primere upotrebe navedenih operatora.

```
> not (1 + 1 /= 2) && 2 + 2 <= 5 || 1^0 == 0^1
True : Bool
> 2^6 - 0x100 / 4 * (1 + 2)
-128 : Float
> "Spojen " ++ "string!" == "Spojen string!"
True : Bool
```

Listing 5: Primeri upotrebe osnovnih operatora

## Funkcije

Sintaksa za definisanje funkcija je veoma jednostavna i prikazana je u listingu 6.

```
{-
  nazivFunkcije param1 param2 ... =
    izraz
-}
deljivSa x y =
  modBy x y == 0

dobarDan x = "Dobar dan, " ++ x ++ "!"
```

Listing 6: Primeri definisanja funkcija

Ime funkcije obavezno počinje malim slovom, nakon čega sledi niz slova (velikih i malih), simbola `_` i brojeva. Po konvenciji, sva slova se navode u neprekidnoj sekvenci, stoga je preporučena kamilja notacija (`camelCase`). Parametri se odvajaju razmakom, dok se zagrade ne navode ni prilikom definisanja, ni pozivanja funkcije. Ipak, primena funkcije je levo asocijativna, pa je česta upotreba zagrada za ograđivanje izraza. Telo funkcije predstavlja jedan jedini izraz koji se izvršava prilikom pozivanja, a izračunata vrednost predstavlja povratnu vrednost funkcije. Ne koriste se vitičaste zagrade, ni naredba `return`. Izraz se, po konvenciji, piše u novom redu, ali je moguće i u istom.

Funkcije u Elmu mogu prihvatati funkcije kao parametre i vraćati funkcije kao povratne vrednosti, što ih čini funkcijama višeg reda. Nije moguće navoditi podrazumevane vrednosti parametara, kao ni preopterećivanje funkcija.

## Konstante

U Elmu ne postoje promenljive, jednom definisani podaci se ne mogu promeniti, ali je moguće definisati konstante. Često se u literaturi definisanje konstanti naziva

imenovanjem vrednosti izraza i ne dovodi se u vezu sa funkcijama, ali se konstante mogu posmartati kao *konstantne funkcije*, koje se izvrše tokom kompilacije. Definišu se kao i funkcije, ali bez parametara (listing 7).

### Anonimne funkcije

Anonimne funkcije se definišu slično kao i regularne, umesto imena navodi se simbol  $\backslash$  koji predstavlja grčko slovo lambda -  $\lambda$ , dok se simboli  $\rightarrow$  koristi umesto znaka pridruživanja (listing 7).

```
> broj3 = 3
3 : number
> (\x y -> x + y) broj3 4
7 : number
```

Listing 7: Primer anonimne funkcije

### Moduli

Moduli se koriste za grupisanje funkcija u logičke jedinice i kreiranje imenskih prostora (eng. *namespace*). Osnovni tipovi, kao i funkcije i operatori nad njima definisani su u **Basics** modulu, koji je podrazumevano uvezen i nalazi se unutar **elm/core** paketa. Svaki modul predstavlja jedanu *.elm* datoteku, koja se mora zvati isto kao i modul, dok ime modula mora počinjati velikim slovom. Za definisanje modula koristi se ključna reč **module** nakon koje sledi ime modula, ključna reč **exposing** i lista funkcija kojima se može pristupiti van modula.

```
module Krug exposing (povrsina, obim)
-- module Krug exposing (...) - otkrivanje svega iz modula
pi = 3.14

povrsina r =
    naKvadrat r * pi

obim r =
    2 * r * pi

naKvadrat x =
    x * x
```

Listing 8: Primer modula

Da bi se modul iz listinga 8 koristio u interpreteru (`elm repl`), prvo je potrebno inicijalizovati elm projekat (`elm init`) i u `src` folderu napraviti `Krug.elm` datoteku sa prikazanim sadržajem. Zatim, u interpreteru naredbom `import` treba uvesti modul. Načini korišćenja funkcija iz modula `Krug` prikazani su u listingu 9.

```
import Krug                                -- Krug.obim Krug.poursina
import Krug as K                          -- K.obim K.poursina

import Krug exposing (obim)                -- obim, Krug.poursina
import Krug exposing (..)                  -- obim, poursina
import Krug as K exposing (poursina)      -- K.obim, poursina
```

Listing 9: Primer korišćenja modula

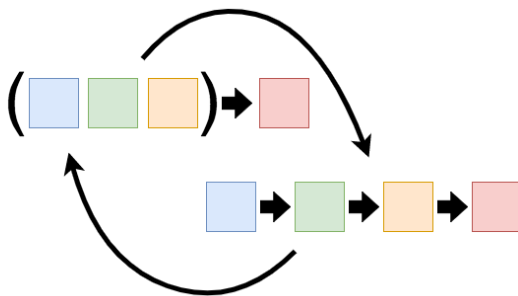
## Tip funkcije

Prilikom definisanje funkcije u interpreteru ili poziva funkcije bez parametara, kao vrednost izraza vraća se `<function>` i tip funkcije. U listing-u 10 prikazano je nekoliko primera tipova funkcija u interpreteru.

```
> not
<function> : Bool -> Bool
> deljivSa
<function> : Int -> Int -> Bool
> \x y z -> x + y + z
<function> : number -> number -> number -> number
> deljivSa 3 --parcijalna primena
<function> : Int -> Bool
```

Listing 10: Tipovi funkcija

U primeru funkcije `not` vidimo da je njen tip `Bool -> Bool`, što je na prvi pogled jasno i znači da se radi o funkciji jednog argumenta koja prihvata vrednost tipa `Bool` i vraća vrednost tipa `Bool`. U slučaju funkcija koje imaju više argumenata, tip funkcije se može posmatrati na način da poslednji tip u nizu koji je razdvojen strelicama (`->`) predstavlja povratni tip funkcije, dok tipovi pre njega predstavljaju tipove argumenata funkcije. Tipovi argumenata se takođe odvajaju strelicama iz razloga što su sve funkcije u Elmu zapravo Karijeve (*Curried*) funkcije, što znači da su sve *n*-arne funkcije zapravo funkcije jednog argumenta koje kao povratnu vrednost imaju funkciju — slika 2.3. Strelica (`->`) je desno asocijativna, a zagrade se izostavljaju zbog jednostavnosti. Na primer, funkcija `deljivSa` ima tip `Int -> (Int -> Bool)`,



Slika 2.3: Regularne i Karijeve funkcije

što znači da prihvata vrednost tip *Int* i vraća funkciju tipa *Int -> Bool*. Karijeve funkcije nam omogućavaju veću fleksibilnost i parcijalnu primenu funkcija, odnosno vezivanje argumenata za konkretne vrednosti (listing 10).

### Anotacija tipa funkcije

Kao što je prethodno prikazano, Elm sam zaključuje tip funkcije, ali dozvoljava i korisniku da sam navede tip u liniji iznad definicije (listing 11).

```
> deljivSa: Int -> Int -> Bool
| deljivSa x y =
|   modBy x y == 0
|
<function> : Int -> Int -> Bool
```

Listing 11: Anotacija tipa funkcije

Korišćenje anotacije tipova nije obavezno, ali je vrlo preporučljivo iz više razloga. Prilikom kompilacije proverava se poklapanje anotacije sa stvarnim tipom funkcije, što dovodi do lakšeg uočavanja i otklanjanja grešaka. Pored toga, anotacije predstavljaju veoma dobar vid dokumentacije, a činjenica da kompilator uvek poredi navedeni i stvarni tip nam garantuje da je dokumentacija uvek važeća.

### Funkcijski operatori

Operatore nad funkcijama možemo podeliti prema tipu, na operatore prosleđivanja i operatore kompozicije funkcija, i prema smeru u kom se primenjuju, unapred ili unazad.

Operatori prosleđivanja ili **pipe** operatori su zapravo operatori primene funkcije i omogućavaju pisanje čitljivijeg koda sa manje zagrada.

- `<|` - pipe operator unazad radi isto što i primene funkcije, s tim što nas oslobađa pisanja zagrada. Preciznije, `f <| x` je drugi zapis za `f (x)`
- `|>` - pipe operator inspirisan je Unix pipe-om, odatle i naziv, i služi za prosleđivanje argumenta funkciji. Preciznije, `x |> f` je drugi zapis za `f (x)`

Operator kompozicije unazad `<<` predstavlja operator matematičke kompozicije funkcija `o`. Tako da se definicija kompozicije dve funkcije,  $(g \circ f)(x) = g(f(x))$ , u Elmu može posmatrati kao: `(g << f) x == (\x_-> g ( f x_)) x`. Operator kompozicije `>>` je simetričan operatoru `<<`, stoga važi: `(f << g) x == (g >> f) x`.

## Osnovne strukture podataka

Osnovne strukture podatka u Elmu čine liste, torke i slogovi.

### Liste

Lista u Elmu predstavlja kolekciju u obliku jednostruko povezane liste. Elementi se navode unutar uglastih zagrada—`[]` i moraju biti istog tipa. Funkcije za rad sa listama nalaze se unutar `List` modula, među kojim su i tri veoma važne funkcije u funkcionalnoj paradigmi — `map`, `filter` i `fold` (ili `reduce` u drugim programskim jezicima). Pored operatora za nadovezivanje `++`, postoji i operator `::` koji dodaje element na početak liste, iz istorijskih razloga ovaj operator naziva se *kons* (eng. *cons*).

```
> "aaa" :: ["bbb", "ccc"]
["aaa", "bbb", "ccc"] : List String
> List.map (List.member 2) [[1,2,3],[2,2],[42]]
[True,True,False] : List Bool
> [1,2]++[3,4,5] |> List.filter (\x -> modBy 2 x == 0) |>List.length
2 : Int
> List.foldl (\x y -> x + y) 0 <| List.range 1 5
15 : Int
```

Listing 12: Primeri lista različitih tipova i funkcija za rad sa njima

### Torke

Za razliku od listi koje mogu imati promenljivi broj elemenata istog tipa, torke predstavljaju kolekcije fiksne dužine čiji elementi ne moraju biti istog tipa. Mogu



sadržati samo dva ili tri elementa, navode se unutar običnih zagrada— (). U torke se ne mogu ubacivati elementi niti se iz torke mogu uklanjati elementi. Funkcije nad torkama koje imaju dva elementa nalaze se u modulu `Tuple`, dok se za rad sa tokrama od tri elementa koristi poklapanje obrazaca o kojem će biti više reči kasnije.

```
> (1,"2",'3')
(1,"2",'3') : ( number, String, Char )
> (1,2) == Tuple.pair 1 2
True : Bool
> Tuple.second ("nebitan", "drugi")
"drugi" : String
> Tuple.mapFirst String.length ("mapiran", 1)
(7,1) : ( Int, number )
```

Listing 13: Primeri torki i upotreba funkcija iz modula `Tuple`

## Slogovi

Slog (eng. *Record*) predstavlja strukuru podataka koja može sadržati više vrednosti različitih tipova, pri čemu je svakoj vrednosti dodeljen naziv. Liče na objekte u JavaScript-u, čak je i sintaksa veoma slična, umesto dvotačke slog koristi znak jednakosti za dodelu naziva. Prilikom definisanja sloga, Elm kreira funkcije za pristup njegovim svojstvima. Nije moguće dodavanje, ni uklanjanje svojstava, ali je dozvoljena promena njihovih vrednosti. Zbog imutabilnosti, ne vrši se promena nad postojećim slogom već se pravi novi.

```
> pera = {ime = "Pera", prezime = "Perić", godine = 23}
{ godine = 23, ime = "Pera", prezime = "Perić" }
  : { godine : number, ime : String, prezime : String }
> {pera | prezime = "Petrović", godine = 24}
{ godine = 24, ime = "Pera", prezime = "Petrović" }
  : { godine : number, ime : String, prezime : String }
> pera.ime
"Pera" : String
> .godine pera
23 : number
> .prezime
<function> : { b | prezime : a } -> a
```

Listing 14: Primeri pristupa i promene svojstava sloga

Pored navedenih struktura podataka, Elm pruža podršku za rad sa nizovima (`Array`), skupovima (`Set`) i rečnicima (`Dict`).

### Tipske promenljive

U listingu 14 vidimo da funkcija za pristup prezimenu ima tip: `{ b | prezime : a } -> a`. Ovo znači da funkcija kao argumenat prima slog, koji može biti bilo kog tipa, ali mora imati svojstvo `prezime`, koje takođe može biti bilo kog tipa, i čiji tip je ujedno povraća tip funkcije. Promenljive `a` i `b` nazivaju se *tipske promenljive*, a prisustvo dve tipske promenljive nam govori da one mogu, ali ne moraju, predstavljati različite tipove. U konkretnom primeru `a` i `b` su uvek različitog tipa, dok u slučaju funkcije `Tuple.pair : a -> b -> ( a, b )` mogu biti istog tipa. Prilikom anotacije tipova mogu se koristiti i duža imena tipskih promenljivih, a pravila imenovanja su ista kao i za funkcije.

Tipske promenljive u Elmu ukazuju na prisustvo **parametarskog polimorfizma**, jedine vrste polimorfizma u ovom jeziku.

### Uslovne tipske promenljive

Za razliku od Haskell-a, sistem tipova u Elmu nije toliko složen i umesto tipskih klasa (eng. *typeclasses*)[4] poseduje jednostavniji koncept — *uslovne tipske promenljive*.

Uslovne tipske promenljive omogućavaju da se na određni način ograniči skup tipova koji se može koristiti u izrazima. Najčešći primer je `number`, koji dozvoljava isključivo `Int` ili `Float` tipove.

U trenutnoj verziji (*0.19.1*) postoje četiri uslovne tipske promenljive:

1. `number` — dozvoljava tipove `Int` i `Float`
2. `appendable` — dozvoljava tipove `String` i `List a`
3. `comparable` — dozvoljava tipove `Int`, `Float`, `Char`, `String`, liste i torke koje sadrže `comparable` vrednosti
4. `compappend` — dozvoljava tipove `String` i `List comparable`

### Operatori i tipske promenljive

```
> (+) 1 2
3 : number
> (*)
<function> : number -> number -> number
> (++)
<function> : appendable -> appendable -> appendable
> (==)
<function> : a -> a -> Bool
> (>=)
<function> : comparable -> comparable -> Bool
```

Listing 15: Upotreba operatora u prefiksnoj notaciji

Operatori u Elmu predstavljaju funkcije koje se mogu pozivati u infiksnoj notaciji. Takođe, mogu se pozivati i u prefiksnoj ukoliko ih navedemo unutar zagrada: (+), (++), (>=)... , dok pozivanjem bez argumenata možemo videti i kog su tipa (listing 15).

U ranijim verzijama jezika bilo je moguće definisati korisničke operatore, ali je ta opcija izbačena u verziji 0.19.0., a od verziji 0.18.0 nije moguće pozivanja binarnih funkcija u infiksnoj notaciji.

## Alijasi tipova

Prilikom definisanja funkcija koje rade nad istim strukturama podataka višestruko navodimo iste anotacije tipova, pritom anotacije podataka mogu biti predugačke, samim tim i teško čitljive. Alijasi tipova nam omogućavaju ponovnu upotrebu anotacija i bolju čitljivost. Definišu se pomoću ključnih reči `type alias`, nakon kojih sledi ime koje mora počinjati velikim slovom.

```
>type alias MatricaInt3 = List (List (List Int))
>
>type alias Osoba = {ime : String, prezime : String, godine : Int }
> Osoba
<function> : String -> String -> Int -> Osoba
> Osoba "Pera" "Perić" 23
{ godine = 23, ime = "Pera", prezime = "Perić" } : Osoba
```

Listing 16: Primeri definisanja alijasa tipova

Prilikom kreiranja alijasa za slog kreira se i konstruktor za slog, što se može videti u listingu 16. Redosled argumenata funkcije za konstrukciju identičan je redosledu u alijasu.

## Korisnički definisani tipovi

Pored korišćenja alijasa za postojeće tipove podataka, Elm pruža mogućnost kreiranja novih tipova. Korisnički definisani tipovi se često nazivaju i *unijski tipovi*, jer mogu predstavljati uniju više varijanti definisanog tipa. Definišu se ključnom reči `type`, a varijante se odvajaju simbolom `|`.

Slično kao kod alijasa tipova za slogove i ovde se kreiraju konstruktori za definisani tip.

```
> type VectorF4 = Vector4F Float Float Float Float
> Vector4F
<function> : Float -> Float -> Float -> Float -> VectorF4
>
> type StatusPrijava
  = NaCekanju
  | Greska String
  | Uspesno {id : Int, token : String}
> NaCekanju
NaCekanju : StatusPrijava
> Greska
<function> : String -> StatusPrijava
> Uspesno
<function> : { id : Int, token : String } -> StatusPrijava
```

Listing 17: Primeri korisnički definisanih tipova

## Kontorla toka

U Elmu nema izvršavanja naredbi, već samo evaluacije izraza, tako da umesto naredbi grananja imamo izraze `if` i `case`, a rekurziju umesto petlji.

### Izraz `if`

Izraz `if` se može posmatrati kao ternarni operator u JavaScript-u, C++-u i mnogim drugim programskim jezicima.

Ključna reč `else` je sastavni deo izraza `if` tako da `else` „grana” uvek postoji. Mogu se navoditi i ugnježdeni izrazi `if`, pa `else if` grana predstavlja korišćenje izraza `if` nakon ključna reč `else`.

```
{-      uslov  ?  izraz1  :  izraz2  - ternarni operator
      if uslov then izraz1 else izraz2  - if izraz
-}
if x >= 0 then "pozitivan" else "negativan"
if x == 1 then x * 2 else if x == 2 then x / 2 else x
```

Listing 18: Sintaksa izraz if i primer upotrebe

### Izraz case

Izraz `case` predstavlja pandan `switch` naredbi u drugim programski jezicima. Prilikom korišćenja izraza `case` moraju se pokriti sve mogućnosti, ukoliko to nije slučaj kompilator prijavljuje grešku. Za podrazumevani slučaj može se koristiti simbol `_`. Izraz `case` zauzima značajno mesto u Elmu, jer se koriste u **poklapanju obrazaca**.

```
case mesto of
  1 -> "zlato"
  2 -> "srebro"
  3 -> "bronza"
  _ -> "zahvalnica"
```

Listing 19: Primer upotrebe izraza case

### Izraz let

Budući da ne postoje blokovi naredbi, izrazi `let` nam omogućavaju da ograničimo oblast važenja — dosega (eng. *scope*) funkcija i konstanti u okviru jedne funkcije. Doprinose boljoj čitljivosti koda, a moguće je koristiti anotacije tipova unutar njih.

```
let
  nula: Int
  nula = 0

  pozitivan: Int -> Bool
  pozitivan =
    \x -> x > nula
in pozitivan 10
```

Listing 20: Primer upotrebe izraza let

## Rekurzija

Rekurzivno definisana funkcija poziva samu sebe, čime se postiže ponavljanje izvršavanja koje se u imperativnom programiranju ostvaruje korišćenjem petlji. U mnogim slučajevima, umesto rekurzije mogu se koristiti funkcije nad listama.

```
factorial n =  
  if n <= 1 then 1  
  else n * factorial (n - 1)  
  
factorialFold n =  
  List.foldl (*) 1 (List.range 1 n)
```

Listing 21: Upotreba rekurzije i foldl funkcije za iteraciju kroz listu

## Poklapanje obrazaca

Poklapanje obrazaca može se posmatrati kao pokušavanje usklađivanja (poklapanja) ulaznog podatka sa unapred definisanim obrascem. Ukoliko dođe do usklađivanja, poklopljenim vrednostima se može prisupiti putem identifikatora definisanim u obrascu. Pored spomenutih `case` izraza, u Elmu se poklapanje obrazaca može koristiti u vidu razlaganja slogova ili torki prilikom definisanja funkcija ili korišćenja `let` izraza.

Unutar `case` izraza sekvencijalno se vrši poklapanje obrazaca. Kada dođe do poklapanja izračunava se izraz dodeljen datom obrascu, ne nastavlja se sa poklapanjem. Kompilator prepoznaje ukoliko može doći do nepoklapanja nijednog obrasca i prijavljuje grešku. Takođe, greška se prijavljuje i ukoliko se navede redundantan obrazac, tj. obrazac čiji je skup vrednosti zapravo podskup skup vrednosti prethodno definisanog obrasca. Simbol `_` služi za poklapanje vrednosti koje se ne koriste, a ključna reč `as` se može koristiti ukoliko je potrebno pristupiti celom ulaznom podatku. U listingu 22 mogu se videti primeri poklapanja obrazaca.

## Obrada grešaka pomoću Maybe i Result

U Elmu ne postoje `try catch` blokovi, kao ni `undefined`, `null`, `nil` i ostale slične vrednost prisutne u drugim programskim jezicima. Umesto njih koriste se `Maybe` i `Result` koji potiču iz Haskell-a, gde imamo `Maybe` i `Either`.

```
-- liste
case lista of
  [] -> "prazna lista"
  [_] -> "jedan element"
  [a,b] -> "dva elementa:" ++ a ++ " i " ++ b
  a :: _ -> "više od dve elemenata, prvi je: " ++ a

-- unijski tipovi
case prijava of
  NaCekanju -> "Molimo za strpljive"
  Greska poruka -> "Došlo je do grške: " ++ poruka
  Uspesno {id} -> "Uspešna prijava, id: " ++ String.fromInt id

-- torke
case tacka3D of
  (0, 0, 0) -> "centar"
  (0, _, _) -> "na x-osi"
  _ -> "van x-ose"

-- razlaganje
let
  (x,_,_) = tacka3D
in "x koordinata je " ++ String.fromFloat x

--nije moguće poklapanje ugnježđenih slogova
prikaziPodatke ({ime, adresa} as osoba) =
  ime ++ " " ++ osoba.prezime ++ " " ++ adresa.ulica
```

Listing 22: Primeri poklapanja obrazaca

## Maybe

U slučaju da je potrebno napisati funkciju koja vraća prvi element liste, ukoliko on postoji, rezultat bi bio **baš** prvi element. Dok u slučaju prazne liste, funkcija ne bi vratila **ništa**. Upravo tako radi funkcija `List.head : List a -> Maybe a`, ukoliko se pozove **možda** vrati prvi element.

`Maybe` se definiše kao `type Maybe a = Just a | Nothing` i može se koristiti za opcione argument, obradu grešaka i u slogovima sa opcionim svojstvima. Zapravo svuda gde očekivani podatak može, ali ne mora, postojati.

## Result

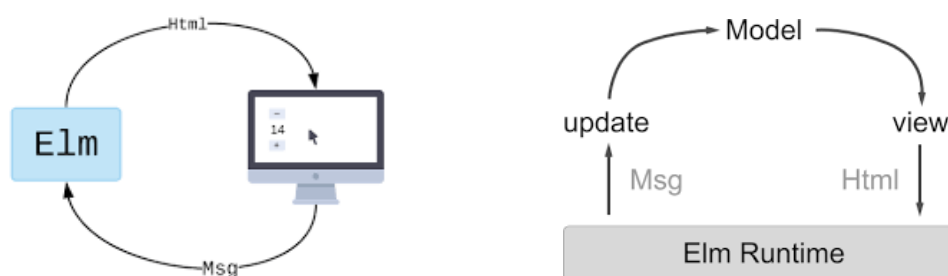
Za razliku od `Maybe`, koji bi u slučaju greške vratio `Nothing`, `Result` nam daje mogućnost pružanja dodatnih informacija o grešci. Definiše se na sledeći način:

```
type Result error value
  = Ok value
  | Err error
```

## 2.5 Arhitektura Elm

Arhitektura Elm je veoma jednostavan obrazac projektovanja veb aplikacija koji se pojavljuje ubrzo nakon nastanka samog jezika Elm. Nastaje prirodno, tako što rani Elm programeri uočavaju iste obrasce u svom kodu, koji se mogu podeliti u tri dela:

1. Model - stanje aplikacije
2. Pogled (eng. *View*) - transformacija stanja u HTML
3. Ažuriranje (eng. *Update*) - promena stanja



Slika 2.4: Elm arhitektura

U levom delu slike 2.4 prikazano je kako radi svaki Elm program: generiše se HTML koji se prikazuje u pretraživaču, nakon čega pretraživač šalje poruku programu ukoliko se nešto dogodilo. U desnom delu slike vidimo šta se dešava unutar programa, na osnovu primljene poruke funkcija `update` kreira novi `model`, koji se prosleđuje funkciji `view`, na osnovu koje se generiše HTML.



Ovaj obrazac projektovanja ce često naziva i MVU (eng. *Model-View-Update*). Za razliku od obrazaca MVC [8] (eng. *Model-View-Controller*) i MVVM [10] (eng. *Model-View-ViewModel*), koji stanje aplikacije dele na više manjih modela, u Elm arhitekturi celokupno stanje aplikacije se nalazi na jednom mestu, tj. modelu, a protok podataka kroz aplikaciju je uvek u jednom smeru.

Arhitektura Elm uticala je na nastajanje velikog broja biblioteka za upravljanje stanja veb aplikacija, među kojima su najpoznatije Redux[14] i Vuex[16]. Takođe, podrška za rad sa MVU obrascem počinje da se pojavljuje i u drugim tehnologijama, jedna od njih je i .NET 5 [9].

## Generisanje HTML sadržaja

Za razliku od drugih radnih okruženja koji uglavnom koriste HTML šablone, Elm za opisivanje izgleda stanice koristi funkcije. Stoga imamo funkcije za kreiranje HTML čvorova i atributa.

```
-- <button id="btn-ok" class="btn-default"> Ok </button>
node "button" [id "btn-ok", class "btn-default"] [text "Ok"]
-- korišćenje pomoćne funkcije button umesto node "button"
  button [id "btn-ok", class "btn-default"] [text "Ok"]
```

Listing 23: Primeri kreiranje HTML čvorova

Funkcija `node` iz modula `Html` predstavlja generičku funkciju za kreiranje HTML čvorova, koja kao argumente prima HTML oznaku, listu atributa i listu čvorova (deca čvora). U listingu 23 predstavljena je analogija kreiranja HTML čvora funkcijom `node` i HTML sintakse. Modul `Html` pruža veliki broj pomoćnih funkcija, koje imaju nazive po HTML oznakama (npr. `button` — listing 23) i omogućavaju bolju čitljivost. Stoga se funkcija `node` koristi prilikom upotrebe korisnički definisanih elemenata ili ukoliko ne postoji pomoćna funkcija za željeni element. Funkcija `text` služi za postavljanje teksta u DOM, dok su `id` i `class` funkcije za kreiranje atributa iz modula `Html.Attributes`.

## Elm program

Prilikom inicijalizacije Elm projekta se, pored paketa `elm/core` sa osnovnim funkcionalnostima i strukturama podataka i paketa `elm/html` za generisanje HTML stranica, nalazi paket `elm/browser` za kreiranje Elm programa u pretraživaču. Unu-

tar ovog paketa, u modulu **Browser** se nalaze funkcije koje nam omogućavaju kreiranje različitih tipova programa, među kojima je i funkcija **sandbox** namenjena za učenje osnova Elm arhitekture i omogućava bazičnu interakciju sa korisnicima, bez komunikacije sa spoljnim svetom. Primer jednostavanog Elm programa upotrebom **sandbox** funkcije prikazan je u listingu 24.

Da bi Elm okruženje znalo koja je polazna tačka programa potrebno je da se u glavnom modulu definiše i izloži konstanta **main**, dok naziv modula nije bitan. Pored funkcija iz modula **Browser**, u slučaju statičkih stranica, konstanta **main** se može definisati funkcijama iz modula **Html**, na primer:

```
main = text "Zdravo svete!"
```

Da bi se pokrenuo program potrebno je pozicionirati se u komandnoj liniji na inicijalizovani Elm projekat i pokrenuti **elm reactor**. Nakon toga, potrebno je otvoriti pretraživač na adresi *http://localhost:8000* i unutar **src** direktorijuma kliknuti na **Main.elm** čime se vrši kompilacija i pokretanje programa. Takođe, primer se može naći i na zvaničnom Elm vodiču[13].

U primeru jednostavnog brojača iz listinga 24 vidimo da funkcija **sandbox** kao argument prima slog sa inicijalnom vrednošću modela i funkcijama za ažuriranje i prikazivanje modela. Program počinje sa pozivanjem funkcije **view** sa parametrom **init**. Unutar funkcije **view** se pomoću funkcije za kreiranje atributa iz modula **Html.Events** mogu definisati načini slanja poruka, a kao rezultat izvršavanja nastaje virtualni DOM, na osnovu kog Elm okruženje izmenjuje stvarni DOM. Ukoliko dođe do odgovarajuće akcije korisnika, u ovom sličaju klikom na dugme, Elm okruženje generiše poruku i prosleđuje je zajedno sa modelom funkciji **update** koja kreira novi model nad kojim se poziva ponovo funkcija **view**. Elm okruženje poredi prethodni virtualni DOM sa novim i vrši minimalan broj izmena.

```
module Main exposing (main)

import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
    Browser.sandbox { init = init, update = update, view = view }

--MODEL
type alias Model = Int

init : Model
init =
    0

-- UPDATE
type Msg
    = Increment
    | Decrement

update : Msg -> Model -> Model
update msg model =
    case msg of
        Increment ->
            model + 1

        Decrement ->
            model - 1

-- VIEW
view : Model -> Html Msg
view model =
    div []
        [ button [ onClick Decrement ] [ text "-" ]
        , div [] [ text (String.fromInt model) ]
        , button [ onClick Increment ] [ text "+" ]
        ]
```

Listing 24: Primer Elm programa

## Glava 3

# Elixir

Sa zvanične stranice Elixir-a [17]:

Elixir je dinamički tipiziran, funkcionalni programski jezik dizajniran za izgradnju skalabilnih i održivih aplikacija.

Elixir koristi Erlang virtualnu mašinu, poznatu po podršci za rad sistema sa malim kašnjenjem, distribuiranim sistemima i sistemima otpornim na greške, takođe se uspešno koristi u razvoju veba, u sistemima sa ugrađenim računarom (eng. *embedded software*), učitavanju podataka (eng. *data ingestion*) i u domenima za obradu multimedije.

Radeći kao *Ruby* programer, autor jezika, Žozé Valim (José Valim) uvideo je probleme rada *Ruby on Rails* okruženja na višejezgarnim sistemima, kao i prednosti funkcionalnog programiranja i Erlang virtualne mašine. Stoga se odlučuje da kreira programski jezik Elixir i prvu verziju jezika objavljuje 2011. godine. Najveći uticaj na razvoj Elixir imali su programski jezici Erlang, sa kojim je semantički vrlo sličan, i Ruby u smislu sintakse.

Ruby je dinamički tipiziran programski, jezik opšte namene koji pruža mogućnost rada u više programskih paradigmi. Nastao je sredinom devedesetih godina prošlog veka, a razvio ga je japanski naučnik i programer Jukihiro Macumoto (Yukihiro „Matz” Matsumoto). Ruby je dizajniran kao programski jezik fokusiran na programera, tako da svojom lakoćom korišćenja, jednostavnošću i fleksibilnošću učini programiranje prijatnijim. Sam autor kaže da pokušava da napravi Ruby što prirodnijim.

Budući da je Elixir nastao na ideji upotrebe Erlang virtualna mašine, više o Erlang-u biće rečeno u sledećem poglavlju. Takođe, pored dva navedena programska

jezika, uticaj na razvoj Elixir-a imali su Cloujer, Haskell i Python.

## 3.1 Erlang

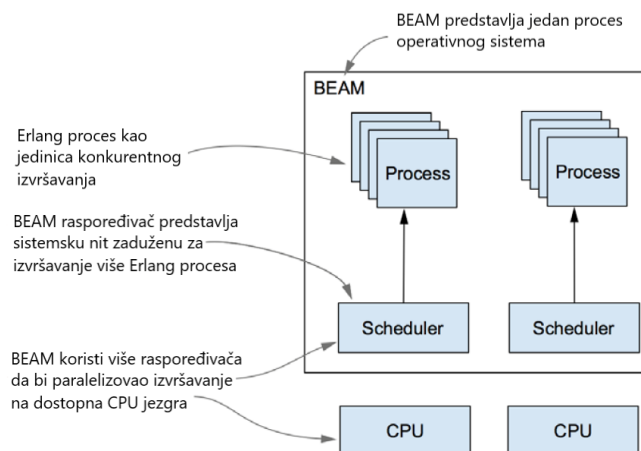
Erlang nije samo programski jezik, već i razvojna platforma za izgradnju skalabilnih i pouzdanih sistema koji neprestano pružaju usluge sa malo ili bez prekida rada. Osmišljena je sredinom osamdesetih godina proslog veka, od strane švedske telekomunikacione kompanije Erikson (eng. *Ericsson*). Da bi Erlang mogao da upravlja telekomunikacionim sistemima kompanije, morao je da bude pouzdan, skalabilan, da ima brz odziv i bude konstanto dostupan, jer telefonska mreža mora da funkcioniše bez obzira na broj istovremenih poziva, neočekivane greške ili hardver. Iako je prvobitno izgrađen za telekomunikacione sisteme, Erlang ni na koji način nije specijalizovan za ovaj domen, ne sadrži podršku za programiranje telefona i drugih telekomunikacionih uređaja. Zapravo, to je razvojna platforma koja pruža posebnu podršku tehničkim, nefunkcionalnim zahtevima kao što su: paralelnost, skalabilnost, tolerancija na greške, distribuiranost i velika dostupnost. U vreme nastajanja Erlanga, programi su se uglavnom koristili bez komunikacije sa nekim veb servisom (eng. *desktop-based software*), pa je upotreba Erlang-a bila ograničena na telekomunikacione sisteme. Međutim, zahtevi modernih sistema i aplikacija poklapaju se nefunkcionalnim zahtevima za koje Erlang pruža podršku, tako da je u poslednje vreme privukao veću pažnju. Erlang pokreće razne velike sisteme, među kojima su aplikacije *WhatsApp* i *WeChat*, *Riak* distribuirana baza podataka i *Heroku Cloud*.

Kao razvojna platforma, Erlang se sastoji od jezika, virtualne mašine, razvojnog okvira (eng. *framework*) i alata.

### Erlang virtualna mašina — BEAM

Jezik Erlang predstavlja primaran način pisanja koda koji se izvršava na Erlang virtualnoj mašini — BEAM. BEAM je skraćenica za *Bogdan's Erlang Abstract Machine*, po Bogumilu „Bogdanu” Hausmanu koji je kreirao originalnu verziju, ali se može posmatrati i kao *Björn's Erlang Abstract Machine*, po Bjornu Gustavsonu koji održava trenutnu verziju virtualne mašine. Kôd napisan u Erlang-u se prevodi u binarni kôd (eng. *bytecode*) koji se unutar BEAM-a izvršava paralelno u vidu veoma lakih *Erlang procesa*. BEAM, umesto da se oslanja na procese i niti operativnog sistema, samostalno raspoređuje konkurentne *Erlang procese* na dostupna procesorska

jezgra, što je prikazano na slici 3.1. Ovi laki procesi su međusobno potpuno izolova-



Slika 3.1: Način izvršavanja Erlang procesa unutar BEAM-a [17]

ni, ne dele memoriju i komuniciraju preko asinhronih poruka, što pruža mogućnost Erlang sistemima da budu skalabilni, distribuirani i otporni na greške.

## OTP razvojni okvir

OTP je skraćenica za *Open Telecom Platform*, što je donekle pogrešan naziv, jer nije toliko povezana sa telekom sistemima i predstavlja razvojni okvir opšte namene koji apstrahuje tipične zadatke Erlang sistema:

- Obrasce konkurentnosti i distribiranosti
- Detekciju i oporavak od greški u konkurentim sistemima
- Pravljanje biblioteka
- Postavljanje (eng. *deployment*) sistema
- Uživo ažuriranje softvera

OTP predstavlja sastavni deo Erlang-a, stoga i zvanična distribucija nosi naziv Erlang/OTP.

## Erlang i Elixir

Elixir predstavlja alternativni načina pisanja programa koji se izvršava na Erlang virtualnoj mašini i samim tim preuzima sve osobine Erlang platforme. Za razliku od

Erlang-a pruža dodatne koncepte koji omogućavaju značajno smanjenje ponavljajućeg i šablonskog (eng. *boilerplate*) koda. Elixir i Erlang su kompatibilni, pa Elixir može direktno da koristi Erlang biblioteke i module, važi i obrnuto, a sve što je moguće implementirati u Erlang-u, moguće je i u Elixir-u bez razlike u performansama. Takođe, važno je napomenuti da oba jezika odlikuje nepromenljivost podataka (imutabilnost).

## 3.2 Osnovne karakteristike jezika Elixir

Pored prethodno spomenutih karakteristika preuzetih od Erlang-a, dinamičke tipiziranosti i sintakse slične Ruby-ju, Elixir odlikuje izraženo poklapanje obrazaca, polimorfizam, makroi i metaprogramiranje, podrška za lenjo izračunavanje, kao i to da nije čist funkcionalni jezik.

## 3.3 Uputstvo za instalaciju

Proces instalacije je vrlo jednostavan, potrebno je pratiti uputstva sa zvanične Elixir stranice [17]. Provera uspešne instalacije može se izvršiti pokretanjem komande `elixir --version` u komandnoj liniji ili pokretanjem interaktivnog Elixir-a komandom `iex`.

## 3.4 Osnovni tipovi podataka

Osnovni tipovi podataka u Elixir-u predstavljaju brojevi (celobrojni i realni), buleanske vrednosti, atomi i stringovi. Takođe, u osnovne tipove Elixir ubraja liste i torke. Pregled osnovnih tipova dat je u listingu 25, gde se može videti i način pisanja komentara. Elixir podržava samo linijske komentare i označavaju se simbolom `#`.

```
iex> 1           # integer
iex> 1.0         # float
iex> true        # boolean
iex> :atom       # atom
iex> "elixir"    # string
iex> [1, 2, 3]   # lista
iex> {1, 2, 3}   # torka
```

Listing 25: Pregled osnovnih tipova podataka u Elixir-u

## Celi brojevi

Za razliku od Elm-a, celi brojevi u Elixir-u nemaju ograničen opseg vrednosti, takođe pored heksadecimalnog zapisa, Elixir pruža mogućnost navođenja celobrojnih vrednosti u binarnom i oktalnom obliku. Simbol `_` može se koristiti za odvajanje cifara i kod celih i kod brojeva u pokretnom zarezu (listing 26).

## Brojevi u pokretnom zarezu

Brojevi u pokretnom zarezu su dvostruke preciznosti (64 bita) po standardu *IEEE-754*. Vrednosti se mogu navoditi i u eksponencijalnom zapisu, a za razliku od Elm-a ne postoje vrednost kao što su `NaN` ili `Infinity`.

```
iex> 10_000
10000
iex> 0o77
63
iex> 0b111_111
63
iex> 0xFF
255
iex> 1_000_000.000_123
1000000.000123
iex> 1.23e-3
0.00123
```

Listing 26: Primeri celih i realnih brojeva u Elixir-u

## Atomi

Atom je konstanta čiju vrednost predstavlja njeno ime. Definiše se dvotačkom (`:`), nakon koje slede alfanumerički karakteri, simboli `@` ili `_`, a može se završavati znakom pitanja ili uzvika. Takođe, moguće je koristite i razmake, tada se sadržaj posle dvotačke mora navesti unutar navodnika. Postoji i notacija atoma bez početne dvotačke, ali u tom slučaju moraju počinjati velikim slovom i takvi atomi se nazivaju *alijasi*. Upotreba atoma prikazana je u listingu 27.

Atom se sastoji od teksta i vrednosti. U toku izvršavanja tekst se čuva u *tabeli atoma*, dok vrednost predstavlja referencu na tekst u tabeli atoma, što omogućava brže poređenje atoma kao i veću uštedu memorije. Obzirom da predstavljaju efikasan način za imenovanje konstanti, atomi su svuda prisutni.



```
iex> :neka_vrednost
:neka_vrednost
iex> "atom sa razmacima"
"atom sa razmacima"
iex> AliasAtom # tokom kompilacije se transformiše u "Elixir.AliasAtom"
AliasAtom
iex> AliasAtom == "Elixir.AliasAtom"
true
```

Listing 27: Primeri atoma u Elixir-u

### Buleanske vrednosti kao atomi

Elixir zapravo nema poseban tip za buleanske vrednosti, već koristi atome `:true` i `:false`, i pruža mogućnost njihovog navođenja bez početne dvotačke. U okviru modula `Kernel`, koji predstavlja pandan modulu `Basics` u Elm-u, Elixir pruža funkcije za proveru tipa, među kojima su `is_atom` i `is_boolean` i čija upotreba je predstavljena u listingu 28.

```
iex> is_atom(true)
true
iex> is_boolean(:true)
true
iex> false == :false
true
```

Listing 28: Predstavljanje booleanskih vrednosti preko atoma

Pored booleanskih vrednosti, jos jedna specifična vrednost predstavljena atomom jeste `:nil`, slična vrednosti `null` u drugim programskim jezicima. Ona se takođe može navoditi bez početne dvotačke.

### Stringovi

Stringovi se navode identično kao u Elm-u, unutar jednostrukih ili trostrukih navodnika. U Elixir-u koriste UTF-8 kodiranje i pružaju mogućnost ugrađivanja izraza — interpolaciju stringova, što se može videti u listingu 29. Modul `Kernel` pruža operator konkatencije (`<>`), a funkcije za rad sa stringovima nalaze se u modulu `String`.

```
iex> "Ćao!"
"Ćao!"
iex> "1+1=#{1+1}"
"1+1=2"
iex> "primer " <> "konkatenacije"
"primer konkatenacije"
iex> String.length("Ćao")
3
```

Listing 29: Primeri stringova u Elixir-u

### Bitstringovi i sekvence bajtova (eng. *binaries*)

Kao i kod buleanskih vrednosti, Elixir nema poseban tip za stringove, već za njihovu reprezentaciju koristi sekvencu bajtova (eng. *binary*) i UTF-8 kodiranje. Zapravo fundamentalni tip za njihovu reprezentaciju jeste **bitstring**, koji predstavlja neprekidni niz bitova u memoriji. Dok *binary* predstavlja bitstring čiji je broj bitova deljiv sa osam. Bitstringovi se navode kao sekvenca brojeva unutar znakova << i >>. Podrazumevano se koristi osam bitova za čuvanje svakog broja u bitstringu, ali je moguće navesti broj bitova pomoću modifikatora `::n` da bi se označila veličina od `n` bitova, ili u opširnijoj notaciji `::size(n)`. U listingu 30 vidimo da je operator

```
iex> is_binary(<<1,2>>)
true
iex> is_bitstring(<<1::1>>) # 1 bit
true
iex> is_binary(<<1::1,2>>) # 9 bitova
false
iex> <<1::1,0::1,1::1>>
<<5::size(3)>>
iex> is_binary("neki string")
true
iex> byte_size("Ćao") # Ć zauzima 2 bajta
4
iex> ?a # određivanje koda karakter a
97
iex> <<97,98,99>>
"abc"
iex> "123" <> <<0>> # konkatenacija
<<49, 50, 51, 0>>
```

Listing 30: Predstavljane stringova kao niz bajtova

<> zapravo operator kontatenacije bitstringova i ukoliko sadržaj bitstringa čine samo ASCII karakteri, interaktivni Elxir ispisuje string. Elixir nema tip `Char` za rad sa karakterima, ali daje mogućnost određivanja koda pomoću simbola `?`, nakon kog se navodi karakter.

## Liste

Kao i u Elm-u, lista kao tip predstavlja jednostuko povezanu listu, čiji se elementi navode unutar uglastih zagrada, ali za ralik u Elm-a ne moraju biti istog tipa. Lista spada u nabrojive (eng. *enumerable*) tipove, pa se pored funkcija iz modula `List`, mogu koristiti i funkcije iz modula `Enum`. Takođe, u modulu `Kernel` se nalaze operatori za proveru da li element pripada listi (`in`), za nadovezivanje (`++`) i oduzimanje (`--`) listi, kao i funkcije za određivanje glave, repa i dužine liste. Umesto operatora `cons`, Elixir pruža sintaksu u obliku `[glava|rep]` koja se može koristiti u kreiranju nove list i poklapanju obrazaca. Primeri upotrebe nekih od navedenih funkcija prikazani su u listingu 31, a nešto više o modulu `Enum` i funkcijama višeg reda biće rečeno kasnije.

```
iex> [1,2,3, "123", true] ++ [:atom, ["aaa", "bbb"]]
[1, 2, 3, "123", true, :atom, ["aaa", "bbb"]]
iex> [1, 2, 1, 3, 5] -- [1, 5, 6]
[2, 1, 3]
iex> "abc" in [1, 2, "abc", false]
true
iex> List.insert_at([0,1,2], 1, 3)
[0, 3, 1, 2]
iex> tl([1,2,3,4])
[2, 3, 4]
iex> Enum.sum([1 | [2 | [3, 4]]])
10
```

Listing 31: Rad sa listama u Elixir-u

## Liste karaktera (eng. *Charlists*)

Lista karaktera je lista celih brojeva, gde svaki element predstavlja jedan karakter. Veoma je slična stringovima, za navođenje se koriste apostrofi umesto navodnika, ali glavna razlika je u internoj reprezentaciji, kao i funkcijama koje se nad njima izvršavaju. Mogu se navoditi u jednostrukoj i trostrukoj notacija, a moguća je i interpolacija. U listingu 32 dati su primeri liste karaktera.

```
iex> '''
Primer u
više redova
'''
'Primer u\nviše redova\n'
iex> '2+2=#{2+2}'
'2+2=4'
iex> [97,98,99]
'abc'
iex> '123' ++ [0]
[49, 50, 51, 0]
```

Listing 32: Primeri lista karaktera u Elixir-u

## Torke

I u Elixir-u, se torke navode unutar vitičastih zagrada i mogu sadržati elemente različitih tipova, ali za razliku od Elm-a, broj elemenata nije ograničen i elementi se mogu uklanjati i dodavati. Iako dozvoljavaju promenljiv broj elemenata, torke su zamišljene kao kolekcija podataka fiksne dužine i ne spadaju u nabrojive tipove. U modulu `Kernel` se nalaze funkcije za pristupanje i ažuriranje elemenata, kao i funkcija za održivanje broja elemenata torke, dok se ostale funkcije za rad sa torkama nalaze u modulu `Tuple`. Prikaz rada navedenih funkcija dat je u listingu 33.

```
iex> tuple_size({1, "aaa", false})
3
iex> elem({:prvi, :drugi, :treći}, 0)
:prvi
iex> put_elem({1, 2}, 1, 4)
{1, 4}
iex> Tuple.insert_at({1, 2}, 1, 4)
{1, 4, 2}
iex> Tuple.to_list({1, 2, 3})
[1, 2, 3]
```

Listing 33: Rad sa torkama u Elixir-u

## Drugi ugrađeni tipovi podataka

Pored navedenih, Elixir poseduje i druge ugrađene tipove podataka:

- Referenca — jedinstven podatak u jednoj BEAM instanci i jedinstvenost je zagarantovana samo tokom životnog veka te instance
- Identifikator procesa (*pid*) — koristi se za identifikaciju Erlang procesa
- Identifikator porta — koristi se za identifikaciju portova, mehanizma koji se koristi za komunikaciju sa spoljnim svetom (datotekama, programima...)

## Asocijativne strukture podataka

Osim navedenih osnovnih tipova, dve vrlo važne asocijativne strukture podataka koje se veoma često koriste u Elixir programima predstavljaju **liste ključnih reči** (eng. *keyword list*) i **mape**.

### Liste ključnih reči

Liste ključnih reči predstavljaju posebnu vrstu listi gde je svaki element dvočlana torka, čiji je priv član (tj. ključna reč) obavezno atom, a drugi može biti bilo kog tipa. Takođe, jedna ključna reč može se navesti više puta. Elixir pruža i jednostavniju sintaksu koja izuzima pisanje vitičastih zagrada. Za rad sa listama ključnih reči mogu se koristiti sve funkcije i operatori kao i sa običnim listama i, dodatno, funkcije iz modula `Keyword` i operator `[]` za prisup po određenoj ključnoj reči. Prikaz nekih od funkcija i operatora dat je u listingu 34.

```
iex> [{:prvi, 1}, {:drugi, 2}, {:treći, 3}]
[prvi: 1, drugi: 2, treći: 3]
iex> Keyword.get([prvi: 1, drugi: 2, treći: 3], :drugi)
2
iex> lista = [prvi: 1, drugi: 2, treći: 3]
[prvi: 1, drugi: 2, treći: 3]
iex> lista[:prvi]
1
iex> [a: 1, b: 2] ++ [c: "3"]
[a: 1, b: 2, c: "3"]
```

Listing 34: Primeri rada sa listama ključnih reči u Elixir-u

### Mape

Za razliku od liste ključnih reči, Mape predstavljaju kolekciju elemenata u obliku *ključ-vrednost* gde oba člana mogu biti proizvoljnog tipa. Liste ključnih reči čuvaju

uređenje, što je osobina koju nemaju mape, ali u slučaju većeg broja elemenata mape pružaju bolju efikasnost. Takođe, ključevi unutar mape moraju biti jedinstveni. Elementi mape navode se koristeći `%{}` sintaksu, a ključ i vrednost se odvajaju znakom `=>`. U slučaju da su svi ključevi atomi, može se koristiti jednostavnija sintaksa kao kod liste ključnih reči. Oba slučaja prikazana su u listingu 35.

```
iex> %{1 => false, "bb" => [1,2], {1,2} => 3}
%{1 => false, {1, 2} => 3, "bb" => [1, 2]}
iex> %{a: 123, b: 'bbb', c: "c"}
%{a: 123, b: 'bbb', c: "c"}
```

Listing 35

Pristup elementima mape moguć je pomoću operatora `[]`, kao i funkcija iz modula `Map`, a ukoliko su svi ključevi mape atomi, dostupna je sintaksa oblika *mapa.ključ*. Pored funkcija iz modula `Map`, nad mapama se mogu koristiti i funkcije iz modula `Enum`, jer priadaju nabrojivim tipovima. Mape predstavljaju dinamičku strukturu u kojoj se mogu dodavati i uklanjati elementi, a moguća je i ažuriranje elemenata. Zbog nepromenljivosti, ne menja se postojeći, već se kreira novi element. Pored funkcija iz modula, dostupna je i sintaksa ažuriranja slična sintaksi ažuriranja slogova u Elm-u i prikazana je u listingu 36.

```
iex> mapa = %{a: 1, b: 2, c: 3}
%{a: 1, b: 2, c: 3}
iex>mapa.b
2
iex> mapa[:c]
3
iex> %{mapa | a: 4}
%{a: 4, b: 2, c: 3}
iex>%{ %{ "a" => true, :b => 0} | :b => false}
%{:b => false, "a" => true}
```

Listing 36: Pristup i ažuriranje elemenata mape

## 3.5 Osnovni operatori

Za razliku od Elm-a, Elixir vrši implicitnu konverziju celih brojeva u realne prilikom primene aritmetičkih operatora `+`, `-` i `*` ukoliko operandi nisu istog tipa, dok se u slučaju operatora `/` ona uvek vrši, tako da je rezultat deljenja uvek realan

broj. Za celobrojno deljenje i izračunavanje ostatka pri celobrojnem deljenju koriste se funkcije `div` i `rem`, koje se mogu koristiti samo nad celim brojevima.

Logički operatori u Elixir-u su specifični po tome što rezultat njihove primene ne mora biti buleanska vrednost, već vrednost bilo kog tipa. Takođe, Elixir ima dvostruke operatore za logičko *i*, *ili* i *ne*, koji se mogu podeliti u dve grupe:

1. `and`, `or` i `not` — prvi operand mora biti buleanska vrednosti
2. `&&`, `||` i `!` — oba operanda mogu biti bilo kog tipa

Operatori čiji operandi mogu biti bilo kog tipa zasnivaju se na konceptu istinitosti, gde se vrednosti `nil` i `false` smatraju za lažne, a sve ostale za istinite. Operatori su lenjo izračunljivi, tako da u slučaju primene operatora konjukcije rezultat predstavlja drugi operand samo ako je prvi istinit ili se može smatrati istinitim u slučaju operatora `&&`. Dok prilikom primene operatora disjunkcije rezultat je prvi operand ukoliko je tačan ili se može smatrati tačnim (operator `||`), a u suprotnom vraća se vrednost drugog operanda. Rad sa logičkim operatorima prikazan je u listingu 37.

```
iex> true and true
true
iex> false or 1
1
iex> not 1
** (ArgumentError) argument error
iex> !1
false
iex> nil && "bilo sta"
nil
iex> :atom || false
:atom
```

Listing 37: Primeri upotrebe logičkih operatora

Operatori poređenja u Elixir-u slični su kao i u drugim jezicima. S tim što pored operatora `==`, `!=`, `<=`, `>=`, `< i >`, postoje i operatori `===` i `!==` koji se od operatora `==` i `!=` razlikuju samo prilikom poređenja brojeva, tako što ne vrše konverziju tipova, tj. vrše poređenje po tipu i vrednosti. Upotreba operatora poređenja prikazana je u listingu 38.

Takođe, u listing 38 vidimo da Elixir dozvoljava poređenje vrednosti različitih tipova i to po sledećem uređenju: `number < atom < reference < function < port < pid < tuple < map < list < bitstring`

```
iex> 1 == 1.0
true
iex> 1 === 1.0
false
iex> 1 == "1"
false
iex> 1 < false and "12" > '123'
true
```

Listing 38: Rad sa operatorima poređenja

## 3.6 Poklapanje obrazaca

U prethodnom poglavlju nije naveden jedan od najvažnijih operatora u programskom jeziku Elixir — operator uparivanja ili poklapanja (`=`), koji ima veoma bitnu ulogu u poklapanju obrazaca. Za razliku od Elm-a gde se poklapanje obrazaca koristi isključivo unutar izraza `case` i `let`, u Elixir-u je ono dosta prisutnije i koristi se prilikom definisanja promenljivih i funkcija, kao i u kontroli toka.

Operator uparivanja pokušava da izraz sa leve strane upari sa izrazom sa desne strane, ukoliko ne dođe do uspešnog uparivanja prijavljuje grešku — `MatchError`. U većini drugih programskih jezika, operator `=` koristi se za dodeljivanje vrednosti promenljivoj, što se u Elixiru-u postiže kroz poklapanje obrazaca. Ukoliko dođe do uspešnog poklapanja, promenljive u levom izrazu *vezuju se* za odgovarajuće vrednosti u desnom. Imenovanje promenljivih slično je imenovanju atoma, s tim što umesto dvotačke promenljiva mora počinjati malim slovom, po konvenciji se koristi *snake\_case* notacija. Moguće je koristiti operator uparivanja bez promenljivih, kao i izvršiti ponovno vezivanje promenljive za drugu vrednost, što je prikazano u listingu 39. Takođe, ukoliko ne želimo ponovno vezivanje možemo koristiti *pin* operator `^`.

```
iex> x = 1
1
iex> 1 = 1
1
iex> x = x + 1
2
iex> 1 = x
** (MatchError) no match of right hand side value: 2
iex> ^x = 1
** (MatchError) no match of right hand side value: 1
```

Listing 39: Vezivanje promenljive za vrednost



Operator uparivanja se često upotrebljava za pristupanje elementima torki, listi i mapa, gde se može izvršiti vezivanje više promenljivih odjednom. Kao i u Elm-u, simbol `_` se može koristiti za poklapanje vrednosti koje se neće dalje koristiti. Primeri poklapanja obrazaca listi i torki dati su u listingu 40.

```
iex> {a, b} = {1, 2}
{1, 2}
iex> [c, c] = [3, 3]
[3, 3]
iex> {1, 2, 3, d} = {a, b, c, {4}}
{1, 2, 3, {4}}
iex> [ _ | rep ] = 'abcd'
iex> rep
'bcd'
```

Listing 40: Poklapanje obrazaca listi i torki

U slučaju mapa moguće je parcijalno poklapanje, tj. leva strana ne mora da sadrži sve ključeve desne, što omogućava pristupanje samo potrebnim vrednostima iz mape. Parcijalno poklapanje prikazano u listingu 41.

```
iex> %{ime: ime, godine: godine} = %{ime: "Pera", godine: 25}
%{godine: 25, ime: "Pera"}
iex> "#{ime} ima #{godine} godina."
"Pera ima 25 godina."
iex> %{ime: ime} = %{ime: "Pera", godine: 25}
%{godine: 25, ime: "Pera"}
iex(55)> %{ime: ime, prezime: prezime} = %{ime: "Pera", godine: 25}
** (MatchError) no match of right hand side value: %{godine: 25, ime: "Pera"}
```

Listing 41: Poklapanje obrazaca mapa

## 3.7 Funkcije

Elixir, kao i Elm, poseduje imenovane i anonimne funkcije. Imenovane funkcije se navode isključivo unutar modula, a sintaksa je veoma jednostavna i prikazana je u listingu 42.

Za razliku od Elm-a, gde funkcija predstavlja izvršavanje jednog izraza, u Elixir-u je moguće izvršiti više njih, a poslednji izraz predstavlja povratnu vrednost. U slučaju jednog izraza unutar funkcije može se koristiti i skraćena sintaksa — funkcija pomnozi u listingu 42, dok u funkciji `saberi_i_ispisi` vidimo da je dozvoljeno

```
defmodule Kalkulator do
  def saberi_i_ispisi(x, y) do
    IO.puts("#{a} + #{b} = #{a+b}") #primer bočnog efekta
    saberi(x, y)
  end

  def pomnozi(x, y), do: x * y

  #privatna funkcija
  defp saberi(x, y) do
    x + y
  end
end
```

Listing 42: Poklapanje obrazaca listi i torki

pisanje funkcija koje imaju bočne efekte. Ukoliko je predviđeno da se neke funkcije koriste samo unutar modula, one se definišu kao privatne ključnom rečju `defp`.

Naziv modula mora počinjati velikim slovom i po konvenciji se pišu u kamilja notaciji, dok je imenovanje funkcija identično imenovanju promenljivih, s tim što funkcija čiji naziv se završava simbolom `?` po konvenciji vraća vrednosti `true` ili `false`, a simbolom `!` se označava funkcija koja može izazvati grešku prilikom izvođenja. Pozivanje imenovanih funkcija moguće je sa i bez navođenja zagrada, dok su argumenti razdvojeni zarezima. Na primer, funkciju `pomnozi` možemo pozvati sa: `Kalkulator.pomnozi(2, 3)` i `Kalkulator.pomnozi 2, 3`.

Elixir, nasuprot Elm-u, nema ugrađene Karijeve funkcije, dozvoljava navođenje podrazumevanih vrednosti, kao i preopterećivanje funkcija, što je i prikazano u listingu 43. Funkciju potpuno određuje modul u kome se nalazi, naziv i arnost (broj argumenata), tako da potpun naziv operatora nadovezivanja listi jeste *Kernel.++/2*.

```
defmodule Pravougaonik do
  def površina(a), do: a * a # Pravougaonik.povrsina/1
  def površina(a, b), do: a * b # Pravougaonik.povrsina/2
end

defmodule Brojac do
  # podrazumevana vrednost za n je 1
  def uvecaj(x, n \\ 1), do: x + n
end
```

Listing 43: Primeri definisanja funkcija

Prilikom definisanja funkcija može se koristiti poklapanje obrazaca za destrukciju argumenata, ali i za preopterećivanje funkcija iste arnosti korišćenjem različitih šablona. Takođe, preopterećivanje funkcije, pored različitog broja argumenata i obrazaca, može se vršiti korišćenjem *čuvara* (eng. *guards*), tj. postavljanjem uslova nad argumentima prilikom definisanja funkcija. U oba slučaja bitan je redosled navođenja, jer se i šabloni i uslovi mogu preklapati. Primer preopterećivanja funkcije upotrebom poklapanja obrazaca i čuvara dat je u listingu 44.

```
defmodule Oblik do
  def površina({:pravougaonik, a, b}), do: a * b
  def površina({:kvadrat, a}), do: a * a
  def površina({:krug, r}), do: r * r * 3.141593
  # podrazumvani slučaj
  def površina(oblik), do: {:error, {:nepoznat_oblik, oblik}}
end

# upotreba čuvara
defmodule Math do
  def sgn(x) when x < 0 do
    -1
  end

  def sgn(0), do: 0

  def sgn(x) when x > 0 do
    1
  end
end
```

Listing 44: Upotrebe čuvara i poklapanja obrazaca prilikom definisanja funkcija

## Anonimne funkcije

Sintaksa navođenja anonimnih funkcija prikazana je u listingu 45. Za razliku od imenovanih funkcija, argumenti se ne navode unutar zagrada po konvenciji, mada je moguće koristiti zagrade. Takođe, prilikom pozivanja anonimnih funkcija obavezno je korišćenje zagrada i navođenje tačke (.) pre njih.

Anonimne (lambda) funkcije u Elixir-u se razlikuju od imenovanih i ubrajaju se u osnovne tipove podataka. Imenovane funkcije nije moguće proslediti kao parametar ili vezati za promenljivu, već isključivo anonimne. Ovaj nedostatak imenovanih

```
iex> saberi = fn x, y ->
...>   x + y
...> end
saber1.(1, 2)
3
```

Listing 45: Primer definisanja i pozivanja anonimne funkcije

funkcija se može prevazići upotrebom operatora `&` (eng. *capture*), koji imenovanu funkciju pretvara u anonimnu. Operator `&` se može koristiti za kraću sintaksu anonimnih funkcija, što je i prikazano u listingu 46.

```
iex> saberi1 = &Kernel.+/2
iex> saberi1.(3, 4)
7
iex> saberi2 = &(&1 + &2) # &n je n-ti argument
iex> saberi2.(5, 6)
11
```

Listing 46: Primer upotrebe `&` operatora

Zatvorenja (eng. *closures*) predstavljaju još jednu specifičnost lambda funkcija u Elixir-u. Prilikom definisanje, lambda funkcija ima pristup svim promenljivim unutar opsega u kom se definiše. Ukoliko koristi promenljive van svog opsega, kreiraju se reference na trenutne vrednost promenljivih i ponovno vezivanje promenljivih ne utiče na izvršavanje funkcije. U listingu 47 dat je primer zatvorenja.

```
iex> x = 3
3
iex> saberi_sa_3 = &(&1 + x)
iex> saberi_sa_3.(3)
6
iex> x = 4
4
iex(35)> saberi_sa_3.(3)
6
```

Listing 47: Primer zatvorenja u Elixir-u

### Operator prosleđivanja - *pipe* operator (`|>`)

Operator `|>` u Elixir-u prosleđuje vrednost sa leve strane kao prvi argument funkciji sa desne strana, što je različito ponašanje koje isti operator ima u Elm-u,

gde se može reći da se vrednost sa leve strane prosleđuje kao poslednji argument funkciji sa desne strane. Takođe, Elixir ne podržava slične operatore koji postoje u Elm-u. Različito ponašanje operatora `|>` u oba jezika predstavljeno je listingu 48.

```
iex> podeli = &(&1 / &2)          > podeli x y = x / y
iex> 1 |> podeli.(2)              > 1 |> podeli 2
0.5                               2 : Float
```

Listing 48: Upotreba **pipe** operatora u Elixir-u (levo) i Elm-u (desno)

## Moduli

Pored funkcija, moduli u Elixir-u mogu sadržati dva veoma važna koncepta — atribute i stukture.

### Atributi

Atributi u modulu se mogu koristiti za definisanje konstanti, takođe Elixir pruža atribute za dokumentovanje koda: `@moduledoc` i `@doc`, čija je upotreba prikazana u listingu 49. Atributi imaju višestruku upotrebu i mnogi bitni koncepti se zasnivaju

```
defmodule Krug do
  @moduledoc "Osnovne funkcije za rad sa krugovima"
  #definisanje konstante
  @pi 3.141593

  @doc "Izračunava obim kruga"
  def obim(r), do: 2*r*@pi

  @doc "Izračunava površinu kruga"
  def površina(r), do: r*r*@pi
end
```

Listing 49: Definisanje konstante i dokumentovanje koda pomoću atributa

na njima, među kojima je i specifikacija tipova (eng. *typespecs*) koja se može koristiti za definisanje korisničkih tipova, kao i za anotaciju tipova sličnu anotaciji u Elm-u. Više o specifikaciji tipova i samim atributima može se videti na zvaničnoj stranici.

## Strukture

Strukture predstavljaju koncept razvijen na osnovu mapa, s tim što imaju podrazumevane vrednosti, a ključevi su isključivo atomi. Sintaksa definisanja struktura data je u listingu 50, mogu se navoditi samo unutar modula i to najviše jedna.

```
defmodule Osoba do
  defstruct ime: "Pera", godine: 25
end

defmodule Osoba1 do
  defstruct [:ime, godine: 25]
end
```

Listing 50: Primeri definisanje struktura

Ukoliko se postavljaju podrazumevane vrednosti svih ključeva, moguće je izostaviti navođenje uglastih zagrada. U suprotnom obavezno je njihovo korišćenje, pri čemu se nenavedenim ključevima podrazumevana vrednost postavlja na `nil`. U slučaju da se nekim ključevima ne dodeljuje podrazumevana vrednost, takvi ključevi se obavezno navode na početku liste. Sintaksa kreiranja modula slična je sintaksi kreiranju mapa — `%NazivModula{}`, u listingu 51 može se videti kreiranje struktura iz listinga 50.

```
iex> %Osoba{}
%Osoba{godine: 25, ime: "Pera"}
iex> %Osoba1{}
%Osoba{godine: 25, ime: nil}
#moguće je specifikacija ključeva tokom kreiranja
iex> %Osoba1{godine: 30}
%Osoba1{godine: 30, ime: nil}
```

Listing 51: Primeri kreiranja struktura

Nad strukturama se mogu koristiti funkcije iz modula `Map`, ali ne i funkcije iz modula `Enum`, kao ni operator `[]`. Za pristup ključevima koristi se isključivo sintaksa *struktura.ključ*, a ažuriranje vrednosti može se izvršiti korišćenjem simbola `|` kao kod mapa. Svaka struktura ima specijalni ključ `__struct__` koja sadrži naziv strukture, odnosno modula.

## Korišćenje funkcija iz drugog modula

Funkcije modula mogu se pozivati kao *NazivModula.naziv\_funkcije*, što često nije praktično jer nazivi modula mogu biti veoma dugački. Stoga Elixir nudi dve direktive za kraći i čitljiviji kôd prilikom upotrebe funkcija iz drugih modula — `import` i `alias`, koje su prikazane u listingu 52.

Upotrebom direktive `import` sve funkcije navedenog modula uključuju se u trenutni modul i mogu se pozivati bez navođenja imena modula, a moguće je ograničiti uključivanje samo određenih funkcija. Sa druge strane, `alias` omogućava korišćenja određenog imena (alijasa) za bilo koji modul i prilikom pozivanja funkcija obavezno je njegovo navođenje.

```
alias Api.Accounts.User, as: AccUser
#indentično kao alias Api.Accounts.User, as: User
alias Api.Accounts.User

# uključivanje samo authenticate/2 funkcija
import Api.Accounts, :only [authenticate: 2]
```

Listing 52: Upotreba direktiva `import` i `alias`

## 3.8 Makroi

Makroi predstavljaju jednu od najvažnijih karakteristika Elixir-a u poređenju sa Erlang-om. Omogućavaju metaprogramiranje, tj. pisanje koda koji generiše kôd, što dovodi do značajnog smanjenja količine šablonskog (eng. *boilerplate*) koda, a samim tim i do vrlo čitljivog i elegantnog koda. Upotrebom makroa se tokom kompilacije vrše transformacije nad kodom, čime se ne utiče na performase izvršavanja. Makroi i metaprogramiranje nisu predmet ovog rada, ali je neophodno razumeti kako makroi funkcionišu, jer su mnoge funkcionalnosti Elixir-a implementirane pomoću njih i njihova upotreba je gotovo neizbežna.

Makroi se definišu unutar modula. Pre upotrebe, budući da se prevode tokom kompilacije, moduli u kojima su definisani moraju biti dostupni, što se obezbeđuje direktivom `require`.

Veoma bitan marko koji se često povezuje sa direktivama jeste marko `use`, koji omogućava ubacivanje spoljne funkcionalnosti (koda) u trenutni modul.

## 3.9 Kontrola toka

U delu o funkcijama prikazano je kako se korišćenjem čuvara i poklapanja obrazaca može vršiti kontrola toka izvršavanja, ali ovakva rešenja nisu uvek adekvatna i jednostavna za upotrebu jer podrazumevaju kreiranje zasebnih funkcija i prosleđivanje neophodnih argumenata. Stoga Elixir omogućava i upotrebu makroa `if`, `unless`, `cond` i `case` za standardan način grananja.

### Upotreba makroa *if* i *unless*

Makro `unless` se zapravo može posmatrati kao `if not`, tako da sve što važi za `if`, važi i za `unless`. Sintaksa upotrebe oba makroa data je u listingu 53.

```
if uslov do
...
else
...
end

unless uslov do
...
else
...
end

#kraći oblik
if uslov, do: ..., else: ...    unless uslov, do: ..., else: ...
```

Listing 53: Sintaksa makroa `if` i `unless`

Kao kod operatora `&&` i `||` uslov se smatra tačnim ukoliko njegova vrednosti nije `nil` ili `false`. Dok za razliku od Elm-a, `else` grana nije obavezna i ukoliko se ne navede, a uslov nije istinit, vrednost izrara je `nil`.

### Upotreba makroa *cond* i *case*

Makroi `cond` i `case` se mogu posmatrati kao grananje oblika `if...else if ... else`, pri čemu se u slučaju makroa `cond` proverava istinitost izraza, a kod makroa `case` koristi poklapanje obrazaca. Sintaksa navedenih makroa data je u listingu 54.

```
cond do
  izraz_1 -> ...
  izraz_2 -> ...
  ...
end

case izraz do
  obrazac_1 -> ...
  obrazac_2 -> ...
  ...
end
```

Listing 54: Sintaksa makroa `cond` i `case`



Bitan je redosled navođenja, jer će se izvršiti blok prvog istinitog izraza(`cond`) ili poklopljenog obrasca (`case`). Povratna vrednost predstavlja vrednost izvršenog bloka, a u slučaju da se ne izvrši nijedan blok izbacuje se grešaka.

## Iteracija

Slično Elm-u, Elixir umesto petlji koristi rekurziju i funkcije modula `Enum`, s tim što Elixir pruža mogućnost lenjog izračunavanja kao i upotebu *ubrajanja* (eng. *comprehensions*).

## Modul Enum

Elixir pruža mogućnost rada sa nabrojivim podacima koristeći modul `Enum`, koji sadrži veliki broj funkcija za filtriranje, sortiranje, transformisanje i mnogim drugim uobičajenim operacijama za ovaj tip podataka. Prethodno su navedene liste i mape kao nabrojni tipovi, pored njih Elixir omogućava i rad sa *opsezima* (eng. *ranges*), čija se upotreba korišćenjem nekih od funkcija iz modula `Enum` može videti u listingu 55, zajedno sa listama i mapama. Funkcije iz modula `Enum` mogu raditi sa bilo kojim tipom koji implementira `Enumerable` protokol. Više o protokolima biće kasnije u tekstu.

```
iex> Enum.map([1, 2, 3], fn x -> x * 3 end)
[3, 6, 9]
iex> Enum.reduce(%{1 => 2, 3 => 4}, 0, fn {k, v}, s -> s + k + v end)
10
iex> Enum.filter(1..10, fn x -> rem(x, 3) == 0 end)
[3, 6, 9]
```

Listing 55: Upotreba funkcija iz modula `Enum` nad listama, mapama i opsezima

## Tokovi (eng. *Streams*)

Tokovi predstavljaju posebnu vrstu nabrojivih tipova koji se mogu koristiti za kreiranje lenjih kompozitnih operacija nad nabrojivim podacima. Tokovi su definisani u modulu `Stream` i funkcije unutar ovog modula izgledaju vrlo slično funkcijama u modulu `Enum`, s tim da je povratna vrednost ovih funkcija uvek tok, što omogućava njihovu kompoziciju. Primer lenjog izračunavanja upotrebom tokova dat je u listingu 56.

```
# kompozicija operacija, ne vrši se izračunavanje
iex> stream = 1..1_000_000 |>
...> Stream.filter(&(rem(&1, 3) == 0)) |>
...> Stream.map(&(&1 * 2))
# lenjo izračunavanje - samo za prvih 5
iex> stream |> Enum.take(5)
[6, 12, 18, 24, 30]
```

Listing 56: Kompozicija tokova i lenjo izračunavanje

## Ubrajanja

Filtriranje i mapiranje nad nabrojivim podacima predstaju čestu pojavu u Elixir programima, te su uvedena *ubrajanja* kao sintaksička olakšica (eng. *syntactic sugar*), koja grupiše ovakve operacije u `for` specijalnu formu. Upotreba ubrajanja prikazana je u listingu 57, a više o njima može se pronaći na zvaničnoj stranici.

```
for n <- 1..5, do: n * n
[1, 4, 9, 16, 25]
iex> for i <- [:a, :b, :c], j <- [1, 2], do: {i, j}
[a: 1, a: 2, b: 1, b: 2, c: 1, c: 2]
iex> nije_paran? = &(rem(&1, 2) != 0)
iex> for n <- 1..10, nije_paran?.(n), do: 2 * n
[2, 6, 10, 14, 18]
```

Listing 57: Primeri upotrebe ubrajanja

## Upravljanje greškama

Elixir za razliku od Elm-a, nema tipove kao `Maybe` i `Result`, ali je praksa da se prilikom pisanja funkcija, koje mogu dovesti do greške, rezultat izvršavanja vraća kao torka koja podseća na `Result` ili `Maybe` u slučaju da nije potrebna poruka o grešci. U listingu 58 može se videti primer takve obrade grešaka. Ipak, u nekim

```
iex> case File.read "primer.txt" do
...>   {:ok, sadrzaj}   -> IO.puts "Ok: #{sadrzaj}"
...>   {:error, razlog} -> IO.puts "Error: #{razlog}"
...> end
```

Listing 58: Primer pravilne obrade grešaka

vrlo retkim slučajevim kada neke biblioteke ili delovi koda nisu implementirani u duhu funkcionalnog programiranja, moguća je obrada grešaka slična većini drugih

programskih jezika — koršćenjem `try`, `catch` i `rescue` mehanizma o kom se može videti više na zvaničnoj dokumentaciji.

## 3.10 Polimorfizam preko protokola

Protokoli predstavljaju mehanizam za postizanje polimorfizma ukoliko je potrebno razlikovati ponašanje u odnosu na tip podatka. Ovo je takođe moguće uraditi korišćenjem poklapanja obrazaca i čuvara, ali protokoli omogućavaju implementiranje različitog ponašanja u različitim delovima koda.

Protokol je zapravo modul koji sadrži samo deklaracije funkcija ali ne i implementaciju, sličan je interfejsima ili apstraktnim klasama u drugim programskim jezicima. Definiše se pomoću makroa `defprotocol`, a primer protokola za izračunavanje veličine podatka dat je u listingu 59. Kada postoji definisan protokol moguće je

```
defprotocol Size do
  def size(data)
end
```

Listing 59: Primer definisanja protokola

dodati neograničen broj implementacija u različitim modulima. Za implementiranje protokola koristi se makro `defimpl` pri čemu se mora navesti protokol koji se implementira, za koji tip i implementacija funkcija. Prilikom implementacije protokola nad strukturama moguće je izostaviti navođenje tipa. Primeri nekoliko implementacij protokola iz listinga može se videti u listingu 60.

```
defimpl Size, for: Map do
  def size(map), do: map_size(map)
end

defmodule User do
  defstruct [:email, :name]

  defimpl Size do
    # dva polja
    def size(%User{}), do: 2
  end
end
```

Listing 60: Primeri implementacije protokola

## Glava 4

# Implementacija MSNR portala

## Glava 5

## Zaključak

# Bibliografija

- [1] Evan Czaplicki. *Elm: Concurrent FRP for Functional GUIs*. 2012. URL: <https://elm-lang.org/assets/papers/concurrent-frp.pdf>.
- [2] Evan Czaplicki. *Github comment*. 2015. URL: <https://github.com/elm/elm-lang.org/issues/408#issuecomment-151656681>.
- [3] *Elm - Veoma brz HTML*. URL: <https://elm-lang.org/news/blazing-fast-html-round-two>.
- [4] Learn You a Haskell - Types **and** Typeclasses. *Miran Lipovača*. URL: <http://learnyouahaskell.com/types-and-typeclasses>.
- [5] *Haskell programski jezik*. URL: <https://www.haskell.org>.
- [6] *Install Elm*. URL: <https://guide.elm-lang.org/install/elm.html>.
- [7] *ML programski jezik*. URL: <https://courses.cs.washington.edu/courses/cse341/04wi/lectures/02-ml-intro.html>.
- [8] *MVC obrazac*. URL: <https://en.wikipedia.org/wiki/Model-view-controller>.
- [9] *MVU obrazac u .NET 5*. URL: <https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui>.
- [10] *MVVM obrazac*. URL: <https://en.wikipedia.org/wiki/Model-view-viewmodel>.
- [11] *Npm Elm*. URL: <https://www.npmjs.com/package/elm>.
- [12] *Obejktni model dokumenta*. URL: [https://en.wikipedia.org/wiki/Document\\_Object\\_Model](https://en.wikipedia.org/wiki/Document_Object_Model).
- [13] *Primer Elm programa*. URL: <https://elm-lang.org/examples/buttons>.
- [14] *Redux biblioteka*. URL: <https://redux.js.org/introduction/prior-art>.
- [15] *Try Elm*. URL: <https://elm-lang.org/try>.

## *BIBLIOGRAFIJA*

---

- [16] *Vuex biblioteka*. URL: <https://vuex.vuejs.org>.
- [17] *zvanična stranica Elixir-a*. URL: <https://https://elixir-lang.org/>.