

UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Milena Dukanac

**JEZIK ELIXIR SA PRIMENOM U
SEKVENCIJIRANJU GENOMA**

master rad

Beograd, 2019.

Mentor:

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

Članovi komisije:

dr Jovana KOVAČEVIĆ, docent
Univerzitet u Beogradu, Matematički fakultet

dr Vesna MARINKOVIĆ, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami i tati

Naslov master rada: Jezik Elixir sa primenom u sekvencioniranju genoma

Rezime:

Ključne reči:

Sadržaj

1	Elixir	1
1.1	Jezici sa najviše uticaja	1
1.2	Osnovne karakteristike	7
1.3	Osnovni tipovi podataka	8
1.4	Osnovni operatori	19
1.5	Poklapanje obrazaca	21
1.6	Imutabilnost podataka	23
1.7	Odlučivanje	24
1.8	Moduli	25
1.9	Direktive	26
2	Sekvencioniranje genoma	28
2.1	Istorija sekvencioniranja genoma	29
2.2	Čitanje celog sekvencioniranog genoma	30
2.3	De novo sekvencioniranje genoma za kratka očitavanja	33
2.4	k-mer counting	34
3	Zaključak	41

Glava 1

Elixir

Elixir je funkcionalan programski jezik nastao 2011. godine. Njegovim tvorcem se smatra **Hosé Valim** (engl. *José Valim*). Elixir je dizajniran za izgradnju skalabilnih i lako održivih aplikacija. Posедуje jednostavnu i modernu sintaksu. Zbog svoje funkcionalne prirode, izuzetnog rada u distribuiranim sistemima i tolerancije na greške koja je na jako visokom nivou, u Elixir-u je rađeno mnogo zanimljivih projekata u vezi sa robotikom. Takođe se uspešno koristi u razvoju veba i u domenu softvera za uređaje sa ugrađenim računarom (engl. *embedded software*).

1.1 Jezici sa najviše uticaja

Na nastanak Elixir-a je uticao programski jezik **Erlang**. Pri njegovom kreiranju značajnu ulogu u smislu sintakse imao je programski jezik **Ruby**, a iz jezika kao što su **Clojure**, **Haskell** i **Python** je preuzeo mnoge koncepte.

Lisp

Lisp je drugi najstariji viši programski jezik koji se i danas veoma koristi. Imao je uticaj na programski jezik Erlang, pa je samim tim uticao i na Elixir. Lisp je projektovao **Džon Makarti** 1958. godine, a prvi put je implementiran od strane **Stiva Rasela** na IBM 704 računar. Rasel je pročitao Makartijev rad i zaključio da izračunavanje u Lispu može da se implementira na mašinskom jeziku. Rezultat toga je bio interpretator koji je mogao da pokreće Lisp programe, ali i da računa vrednosti Lisp izraza. Lisp je zasnovan na matematičkoj teoriji rekurzivnih funkcija (u kojoj se funkcija pojavljuje u sopstvenoj definiciji), a Lisp program je funkcija

koja se primenjuje na podatke. On koristi vrlo jednostavnu notaciju u kojoj se operacije i njihovi operandi prikazuju u zagradi. Na primer, $(+a(*bc))$ predstavlja $a + b * c$. Iako se čini neugodna, ova notacija dobro funkcioniše na računarima. Ime LISP je nastalo od „*LIS*t *PRO*cessor”, a povezane liste su jedan od glavnih tipova podataka. Danas postoji veliki broj dijalekata Lisp-a, a najpoznatiji među njima su **Common Lisp** i **Scheme**. Osnova Lisp-a je funkcionalno programiranje, ali se Lisp zbog raznih drugih svojstava smatra multiparadigmatskim programskim jezikom.

Prolog

Prolog (engl. *PRO*gramming in *LOG*ic) je deklarativan programski jezik namenjen rešavanju zadataka simboličke prirode. On bio osnova za Erlang, te je imao indirektan uticaj i na Elixir. Prolog se temelji na teorijskom modelu logike prvog reda. Početkom 1970-ih godina **Alain Kolmerauer** (engl. *Alain Colmerauer*) i **Filipe Rousel** (engl. *Philippe Roussel*) na Univerzitetu u Marselju (engl. *University of Aix-Marseille*), zajedno sa **Robertom Kovalskim** (engl. *Robert Kowalski*) sa Departmana veštačke inteligencije (engl. *Department of Artificial Intelligence*) na Univerzitetu u Edinburgu (engl. *University of Edinburgh*), razvili su osnovni dizajn jezika Prolog.

Erlang

Firma Erikson je 1981. godine oformila novu laboratoriju **Erikson CSLab** (eng. *The Ericsson CSLab*) sa ciljem da predlaže i stvara nove arhitekture, koncepte i strukture za buduće softverske sisteme. Jedan od projekata nove laboratorije je bio eksperimentisanje sa dodavanjem konkurentnih procesa u programski jezik Prolog i on predstavlja začetak novog programskog jezika. Taj programski jezik je 1987. godine nazvan **Erlang**. Ime je nastalo zahvaljujući inicijativi zaposlenih koji su radili na telefonskim prekidačima, a za koje je jezik dizajniran. Naime, oni su predložili da jezik nosi ime Erlang u čast danskom matematičaru i inženjeru Agneru Krarupu Erlangu (engl. *Agner Krarup Erlang*), a što je ujedno odgovaralo i skraćenici od „**E**ricsson **L**anguage”. Erlang se mogao posmatrati kao dijalekt Prologa sve do 1990. godine. Od tada, on ima svoju sintaksu i postoji kao potpuno samostalan programski jezik. Nakon mnogo godina rada nastajale su sve brže, bolje i stabilnije verzije jezika, kao i **standardna biblioteka OTP** (eng. *The Open Telecom Platform*) (referenca). Erlang i OTP su postali deo slobodnog softvera

(eng. *open source software*) u decembru 1998. godine i mogu se slobodno preuzeti sa zvaničnog sajta jezika Erlang. Veliki broj kompanija danas koristi Erlang u razvoju svojih softverskih rešenja. Neke od njih su: Erikson (engl. *Ericsson*), Motorola, Votsap (eng. *WhatsApp*), Jahu (eng. *Yahoo!*), Fejsbuk (eng. *Facebook*) (referenca).

Erlang je funkcionalan, deklarativan i konkurentan programski jezik. Na planu konkurentnosti je svojevrsan primer. Programski jezik Lisp je prvi predstavio funkcionalnu paradigmu i kao takav je imao veliki uticaj na Erlang. Neki od koncepata i osobina koje je nasledio od Lisp-a su koncept rekurzije, *list comprehension* - sintaktička konstrukcija za kreiranje liste zasnovane na postojećim listama, koja sledi formu matematičke notacije *set-builder* za razliku od upotrebe funkcija mapiranja i filtera. Na početku je Erlang stvaran kao dodatak na Prolog, vremenom je pre-rastao u dijalekt Prologa, a kasnije je zbog svoje kompleksnosti i sveobuhvatnosti evoluirao u potpuno novi programski jezik. Sintaksa jezika Erlang u velikoj meri podseća na sintaksu jezika Prolog (na primer, promenljive moraju počinjati velikim slovom, svaka funkcionalna celina se završava tačkom). Oba jezika u velikoj meri koriste poklapanje obrazaca (eng. *pattern matching*).

Sa druge strane, Erlang je uticao na nastanak programskog jezika Elixir. Uz izmenjenu Erlangovu sintaksu i dopunjenu Erlangovu standardnu biblioteku, Elixir uživa široku popularnost.

Ruby

Ruby je dinamički tipiziran programski jezik otvorenog koda nastao 1995. godine. Fokus kod ovog programskog jezika je na jednostavnosti i produktivnosti. Ruby ima elegantnu sintaksu koja je prirodna za čitanje i lako pisanje. Ruby je takođe i interpretirani programski jezik, što znači da se izvorni kod prevodi u kod razumljiv kompjuteru prilikom svakog izvršavanja programa. Interpretirani programski jezici su sporiji od kompajliranih, ali su fleksibilniji i potrebno je kraće vreme za izradu programa. Međutim, sve više iskusnih Ruby programera se okreće Elixir-u. Zapravo, Elixir je prvi jezik nakon Ruby-ja koji zaista brine o lepoti koda i korisničkom iskustvu vezanom za jezik, biblioteke i ekosistem. Na slici 1.1 su prikazane osobine Ruby-ja koje se direktno preslikavaju u osobine Elixir-a:

Ruby	Elixir
<code>irb</code>	<code>iex</code>
<code>rake tasks</code>	<code>mix</code> (built in)
<code>bundler dependencies</code>	<code>mix</code> (built in)
<code>binding.pry</code>	<code>IEx.pry</code> (built in)
Polymorphism	<u>Protocols</u>
Lazy Enumerables	<u>Streams</u>
Metaprogramming	<u>Macros</u> (used sparingly)
Rails	<u>Phoenix</u>

Slika 1.1: Osobine nasleđene od jezika Ruby

Takođe, na slici 1.2 se može videti i primer koda napisan u oba jezika:

Ruby example (OOP style)

```
class Concat
  attr_reader :value

  def initialize(value)
    @value = value
  end

  def join(another_value)
    value + another_value
  end
end

irb> Concat.new("Yo").join("!!!")
irb> => "Yo!!!"
```

Elixir example

```
defmodule Concat do
  def join(value, another_value) do
    value <> another_value
  end
end

iex> Concat.join("Yo", "!!!")
iex> "Yo!!!"
```

Slika 1.2: Sintaksa jezika Ruby i Elixir

Clojure

Clojure je dinamički tipiziran programski jezik opšte namene nastao 2007. godine. Njegov tvorac je Rič Hiki (engl. *Rich Hickey*). Clojure kombinuje pristupačnost i interaktivni razvoj skriptnog jezika sa efikasnom i robusnom infrastrukturom za višenitno programiranje. On je kompajlirani jezik, ali je i dalje potpuno dinamički tipiziran - svaka funkcija koju podržava Clojure je podržana u toku izvršavanja. Predstavlja dijalekt jezika Lisp i deli njegovu filozofiju *code-as-data* (program je funkcija koja se izvršava nad podacima) i moćan makro sistem. Clojure je pretežno funkcionalni programski jezik i sadrži bogat skup nepromenljivih i postojanih struktura podataka. Elixir je nasledio neke od najboljih Clojure karakteristika - efikasne, nepromenljive strukture podataka (imutabilnost), opcionalno lenjo izračunavanje, protokole.

Python

Python je interpretirani jezik opšte namene čiji tvorac je Guido van Rosum (engl. *Guido van Rossum*). Krajem 1980-ih je koncipiran kao naslednik jezika **ABC**, a prvi put je objavljen 1991. godine. Filozofija dizajna jezika Python naglašava čitljivost koda pridavajući veliki značaj razmaku. Njegove jezičke konstrukcije i objektno-orijentisani pristup imaju za cilj da pomognu programerima da napišu jasan i logičan kod za male i velike projekte. Python je takođe dinamički tipiziran jezik i poseduje sistem za prikupljanje smeća (*garbage collector*). Podržava više paradigmi programiranja uključujući proceduralno, objektno-orijentisano i funkcionalno programiranje. Python interpreteri su dostupni za mnoge operative sisteme. Globalna zajednica programera razvija i održava referentnu implementaciju otvorenog koda **CPython**. Neprofitna organizacija *The Python Software Foundation* upravlja i usmerava resursima za razvoj jezika Python i CPython. Jedna od osobina koje je Elixir nasledio od Python-a je podrška za dokumentaciju u vidu dokumentacionih stringova (engl. *docstrings*) koji omogućavaju povezivanje dokumentacije sa modulima, funkcijama, klasama, metodama.

Haskell

Haskell je čisto funkcionalni jezik koji je statički tipiziran i prilično različit od većine ostalih programskih jezika. Nazvan je po Haskel Bruks Kariju (engl. *Haskell Brooks Curry*), čiji rad u oblasti matematičke logike služi kao osnova za sve funkcionalne jezike. Haskell je zasnovan na lambda računu, pa se stoga lambda koristi kao logo jezika. Nudi kratak, jasan i održiv kod, mali procenat grešaka i veliku pouzdanost. Stoga je pogodan za pisanje velikih softverskih sistema, jer njihovo održavanje čini lakšim i jeftinijim. Jedna od karakteristika koje je Elixir preuzeo od ovog jezika je lenjo izračunavanje.

1.2 Osnovne karakteristike

Tokom 2010. godine José Valim, u to vreme zaposlen na poziciji programera u kompaniji *Platformatec*, radio je na poboljšanju performansi okruženja *Ruby on Rails* na višejezgarnim sistemima.(referenca) 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 virtuelnoj mašini jezika Erlang podstaklo da započne pisanje jezika Elixir. Uticaj projekta na kome je do tada radio odrazio se na to da Elixir ima sintaksu koja je nalik na sintaksu jezika Ruby. Ovaj jezik se pokazao veoma dobro pri upravljanju milionima simultanih konekcija: u 2015. je zabeleženo upravljanje nad 2 miliona *WebSocket* konekcija, dok je u 2017. za skalirani Elixir zabeležena obrada 5 miliona istovremenih korisnika. Elixir se danas koristi u velikim kompanijama, kao što su *Discord* i *Pinterest*.

Elixir je dinamički tipiziran, funkcionalni programski jezik koji se pokreće na virtuelnoj mašini jezika Erlang, pa samim tim i nasleđuje pogodna svojstva koje dolaze sa ovim okruženjem kao što su **konkurentnost** i **tolerisanje grešaka**. Iz tačke gledišta olakšavanja svakodnevnog razvoja sofvera, mnogi koncepti su falili ekosistemu jezika Erlang. Neki od njih su **metaprogramiranje** - tehnika kojom programi imaju mogućnost da druge programe posmatraju kao svoje podatke i na taj način čitaju, pa čak i modifikuju njihov, a samim tim i svoj kod u vreme izvršavanja, **polimorfizam**, **makroi**, **podrška za alate**. Upravo navedene koncepte je Elixir 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.

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

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 pattern matching-a, itd.

1.3 Osnovni tipovi podataka

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

1. Atomi
2. Celi brojevi
3. Brojevi u pokretnom zarezu
4. Portovi
5. Ugrađene torke
6. Liste
7. Mape
8. Funkcije
9. Niske bitova
10. Reference

Svaki od ovih tipova, osim poslednja dva, ima odgovarajuće module koji sadrže funkcije koje se koriste za operacije nad tim tipom. Oni predstavljaju omotač oko primitivnog tipa koji nam omogućava korišćenje dodatnih funkcionalnosti nad njim. U nastavku će biti opisani neki od osnovnih tipova.

Atomi

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

Celi brojevi

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

Brojevi u pokretnom zarezu

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

Liste

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

Elixir koristi uglaste zagrade (`[]`) da označi listu vrednosti. Vredosti mogu biti bilo kog tipa, a primer liste sa vrednostima različitih tipova prikazan je na listingu 1.1:

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

Listing 1.1: Primer liste

Nadovezivanje ili oduzimanje 2 liste korišćenjem operatora `++` i `--` prikazano je na listingu 1.2:

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

Listing 1.2: Nadovezivanje i oduzimanje dve liste

Operatori liste nikada ne menjaju postojeću listu. Rezultat povezivanja listi ili uklanjanja elemenata iz liste je uvek nova lista, jer su strukture podataka u Elixir-u nepromenljive. Jedna od prednosti nepromenljivosti je jasniji kod. Omogućeno je slobodno prosleđivanje podatka sa garancijom da neće biti izmenjeni u memoriji - samo transformisani.

Lista može biti prazna ili se može sastojati od **glave** i **repa**. Glava je prvi element liste, a rep je ostatak liste. Oni se mogu izdvojiti pomoću funkcija `hd/1` i `tl/1`. Dodeljivanje liste promenljivoj, dohvaćanje njene glave i repa prikazano je na listingu 1.3:

```
1 iex(1)> lista = [1, 2, 3, 4]
2 [1, 2, 3, 4]
3 iex(2)>hd(lista)
4 1
5 iex(3)>tl(lista)
6 [2, 3, 4]
```

Listing 1.3: Izdvajanje glave i repa liste

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

Prilikom kreiranja liste može se desiti da je rezultat lista vredosti pod jednostrukim navodnicima. Primer koda koji ilustruje ovo može se videti na listingu 1.4:

```
1 iex(1)>[11, 12, 13]
2 '\v\f\r'
3 iex(2)>[104, 101, 108, 108, 111]
4 'hello'
```

Listing 1.4: Lista vrednosti pod jednostrukim navodnicima

Kada Elixir vidi listu *ASCII* brojeva za štampanje, ispisaće je kao *charlist*-u (doslovno listu znakova). Charlist-e su uobičajene kada se povezuju sa postojećim Erlang kodom. Preuzimanje informacija o vrednosti za koju nismo sigurni kog je tipa može se izvršiti pomoću funkcije `i/1` i može se videti na listingu 1.5:

```
1 iex(1)>i 'hello'
2 Term
3 'hello'
4 Data type
5 List
6 Description
7 This is a list of integers that is printed as sequence of
  characters delimited by single quotes because all the integers
  in it represent valid ASCII characters. Conventionally, such
  lists of integers are referred to as "charlists" (more precisely
  , a charlist is a list of Unicode codepoints, and ASCII is a
  subset of Unicode).
8 Raw representation
9 [104, 101, 108, 108, 111]
10 Reference modules
11 List
12 Implemented protocols
13 Collectable, Enumerable, IEx.Info, Inspect, List.Chars, String.
  Chars
```

Listing 1.5: Preuzimanje informacija o tipu vrednosti

Treba imati na umu da reprezentacije sa jednostrukim i dvostrukim navodnicima u Elixir-u nisu ekvivalentne i da predstavljaju različite tipove. Primer možemo videti na listingu 1.6:

```
1 iex(1)>'hello' == "hello"
2 false
```

Listing 1.6: Dva različita tipa

Torke

Torke se u Elixir-u definišu pomoću vitičastih zagrada `{}`. Kao i liste, mogu sadržati vrednosti bilo kog tipa. Primer torke sa vrednostima različitih tipova i određivanjem njene dužine prikazan je na listingu 1.7:

```
1 iex(1)>{:ok, "hello", 1}
2 {:ok, "hello", 1}
3 iex(2)>tuple_size({:ok, "hello", 1})
4 3
```

Listing 1.7: Primer torke i određivanje njene dužine

Torke su strukture fiksne dužine koje bi trebalo da sadrže svega nekoliko elemenata koji su zapisani u memoriji jedan za drugim. To znači da se pristup elementu torke ili određivanje dužine torke izvršava u konstantnom vremenu. Razlika u odnosu na liste je u semantici upotrebe. Liste se koriste kada se manipuliše kolekcijom, dok se torke, zbog brzine pristupa njihovim elementima, uglavnom koriste za smeštanje povratne vrednosti funkcije. Indeksi torke počinju od nule, a primer se može videti na listingu 1.8:

```
1 iex(1)>tuple = {:ok, "hello", 1}
2 {:ok, "hello", 1}
3 iex(2)>elem(tuple, 1)
4 "hello"
```

Listing 1.8: Izdvajanje elementa torke sa indeksom 1

Takođe je moguće umetnuti novi element na određeno mesto u torki pomoću funkcije *put_elem/3*. Primer koda koji ilustruje upotrebu ove funkcije prikazan je na listingu 1.9:

```
1 iex(1)>tuple = {:ok, "hello", 1}
2 {:ok, "hello", 1}
3 iex(2)>put_elem(tuple, 1, "world")
4 {:ok, "world", 1}
5 iex(3)>tuple
6 {:ok, "hello", 1}
```

Listing 1.9: Umetanje novog elementa u torku

Treba obratiti pažnju da je funkcija *put_elem/3* vratila novu torku. Originalna torka smeštena u promenljivoj *tuple* nije izmenjena. Kao i liste, torke su takođe nepromenljive. Svaka operacija nad torkom vraća novu torku i nikada ne menja postojeću. Ova operacija, kao i operacija ažuriranja torke je skupa, jer zahteva kreiranje nove torke u memoriji. Ovo se odnosi samo na samu torku, a ne na njen sadržaj. Na primer, prilikom ažuriranja torke, svi unosi se dele između stare i nove torke, osim unosa koji je zamenjen. Drugim rečima, torke i liste u Elixir-u mogu da dele svoj sadržaj. Ovo smanjuje količinu memorije koju jezik treba da zauzme i moguće je samo zahvaljujući nepromenljivoj semantici jezika.

Ove karakteristike performansi diktiraju upotrebu struktura podataka. Kao što je već pomenuto, jedan od uobičajenih slučajeva korišćenja torki je prilikom vraćanja dodatnih vrednosti iz funkcije. *File.read/1* je funkcija koja se može koristiti za čitanje sadržaja datoteke. Ona vraća torku, što se može videti na listingu 1.10:

```
1 iex(1)>File.read("C:\elixir\text_document.txt")
2 {:ok, "Hello, world!"}
```

Listing 1.10: Primer korišćenja funkcije *File.read/1*

Ako putanja do fajla postoji, povratna vrednost funkcije je torka sa prvim elementom koji je atom `:ok` i drugim elementom koji je sadržaj datog fajla. U suprotnom, povratna vrednost funkcije će biti torka gde je prvi element atom `:error`, a drugi element opis greške.

Liste ključnih reči i mape

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

Liste ključnih reči

U mnogim funkcionalnim programskim jezicima, uobičajeno je da se koristi lista dvočlanih torki za predstavljanje strukture podataka ključ - vrednost. Lista torki gde je prvi element torke atom (tj. ključ) u Elixir-u se naziva **lista ključnih reči**. Elixir podržava posebnu sintaksu za definisanje takvih lista: `[key : value]`. Zapravo, liste ključnih reči mapiraju liste torki. Primer ovakvog mapiranja prikazan je na listingu 1.11:

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

Listing 1.11: Primer liste ključnih reči

Kako su liste ključnih reči liste, nad njima možemo primenjivati sve operacije dostupne nad listama. Na primer, korišćenjem operatora `++` može se izvršiti dodavanje nove vrednosti listi ključnih reči. Primer koda koji ilustruje ovo dodavanje dat je na listingu 1.12:

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

Listing 1.12: Dodavanje nove vrednosti listi ključnih reči

Elementima liste ključnih reči se pristupa na način prikazan na listingu 1.13:

```
1 iex(1)>lista = [a: 0, a: 1, b: 2, c: 3]
2 [a: 0, a: 1, b: 2, c: 3]
3 iex(2)>lista[:a]
4 0
```

Listing 1.13: Pristup elementu liste ključnih reči

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

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

Elixir obezbeđuje modul koji omogućava manipulisanje listama ključnih reči. Liste ključnih reči su jednostavno liste, i kao takve pružaju iste karakteristike linearnih performansi kao i liste. Što je lista duža, više vremena će biti potrebno za pronalaženje ključa, prebrojavanje elemenata i tako dalje. Iz tog razloga, liste ključnih reči se u Elixir-u koriste uglavnom za prosleđivanje opcionih vrednosti. Za čuvanje mnogo elemenata ili garantovanje pojavljivanja jednog ključa sa maksimalno jednom vrednošću treba koristiti mape.

Mape

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

Mapa se definiše pomoću sintakse `%{}` na način prikazan na listingu 1.14:

```
1 iex(1)> mapa = %{:a => 1, 2 => :b}
2 %{:a => 1, 2 => :b}
3 iex(2)>mapa[:a]
4 1
5 iex(3)>mapa[2]
6 :b
7 iex(4)>mapa[:c]
8 nil
```

Listing 1.14: Primer mape i pristupa njenim elementima

Za razliku od liste ključnih reči, mape su vrlo korisne kod poklapanja obrazaca. Kada se koristi u poklapanju obrazaca, mapa će se uvek podudarati sa poskupom date vrednosti kao što se može videti na listingu 1.15:

```
1 iex(1)> %{ } = %{:a => 1, 2 => :b}
2 %{:a => 1, 2 => :b}
3 iex(2)>%{:a => a} = %{:a => 1, 2 => :b}
4 %{2 => :b, :a => 1}
5 iex(3)>a
6 1
7 iex(4)>%{:c => c} = %{:a => 1, 2 => :b}
8 ** (MatchError) no match of right hand side value: %{1 => :b, :a =>
9   1}
9   (stdlib) erl_eval.erl:453: :erl_eval.expr/5
10   (iex) lib/iex/evaluator.ex:257: IEx.Evaluator.handle_eval/5
11   (iex) lib/iex/evaluator.ex:237: IEx.Evaluator.do_eval/3
12   (iex) lib/iex/evaluator.ex:215: IEx.Evaluator.eval/3
13   (iex) lib/iex/evaluator.ex:103: IEx.Evaluator.loop/1
14   (iex) lib/iex/evaluator.ex:27: IEx.Evaluator.init/4
```

Listing 1.15: Mape pri podudaranju obrazaca

Mapa se podudara sve dok ključevi u obrascu postoje u datoj mapi. Tako, prazna mapa odgovara svim mapama.

Promenljive se mogu koristiti prilikom pristupa, podudaranja i dodavanja ključeva mape, što je dato listingom 1.16:

```
1 iex(1)>mapa = %{a => :jedan}
2 %{a => :jedan}
3 iex(2)>mapa[a]
4 :jedan
5 iex(3)>%{^a => :jedan} = %{1 => :jedan, 2 => :dva, 3 => :tri}
6 %{1 => :jedan, 2 => :dva, 3 => :tri}
```

Listing 1.16: Upotreba promenljivih u mapama

Modul **Map** obezbeđuje razne funkcije za manipulaciju mapama, a neke od njih mogu se videti na listingu 1.17:

```
1 iex(1)>Map.get(%{:a => 1, 2 => :b}, :a)
2 1
3 iex(2)>Map.put(%{:a => 1, 2 => :b}, :c, 3)
4 %{2 => :b, :a => 1, :c => 3}
5 iex(3)>Map.to_list(%{:a => 1, 2 => :b})
6 [{2 => :b}, {:a => 1}]
```

Listing 1.17: Neke od funkcija modula Map

Mape imaju sintaksu za ažuriranje vrednosti ključa prikazanu na listingu 1.18

```
1 iex(1)>mapa = %{:a => 1, 2 => :b}
2 %{2 => :b, :a => 1}
3 iex(2)>%{mapa | 2 => "dva"}
4 %{2 => "dva", :a => 1}
5 iex(3)>%{mapa | :c => 3}
6 ** (KeyError) key :c not found in: %{2 => :b, :a => 1}
7 (stdlib) :maps.update(:c, 3, %{2 => :b, :a => 1})
8 (stdlib) erl_eval.erl:259: anonymous fn/2 in :erl_eval.expr/5
9 (stdlib) lists.erl:1263: :lists.foldl/3
```

Listing 1.18: Ažuriranje vrednosti ključa

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

Ukoliko su svi ključevi u mapi atomi, onda se radi pogodnosti može koristiti sintaksa ključnih reči data listingom 1.19:

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

Listing 1.19: Sintaksa ključnih reči

Još jedno zanimljivo svojstvo mapa je to što obezbeđuju sopstvenu sintaksu za pristup atomskim ključevima. Primer ove sintakse možemo videti na listingu 1.20:

```
1 iex(1)>mapa = %{:a => 1, 2 => :b}
2 %{2 => :b, :a => 1}
3 iex(2)>mapa.a
4 1
5 iex(3)>mapa.c
6 ** (KeyError) key :c not found in: %{2 => :b, :a => 1}
```

Listing 1.20: Sintaksa za pristup atomskim ključevima

Programeri koji programiraju u Elixir-u pri radu sa mapama češće koriste *map.field* sintaksu i poklapanje obrazaca nego funkcije iz modula Map, jer dovode do asertivnog stila programiranja.

Često se koriste mape unutar mapa ili čak liste ključnih reči unutar mapa. Elixir obezbeđuje pogodnosti za manipulisanje ugnježđenim strukturama podataka poput *put_in/2*, *update_in/2* i drugih naredbi koje daju iste pogodnosti koje se mogu pronaći u imperativnim jezicima, a da pritom zadrže nepromenljiva svojstva jezika.

Neka je data struktura prikazana listingom 1.21:

```
1 iex(1)>users = [john: %{name: "John", age: 27, languages:
2 ["Erlang", "Ruby", "Elixir"]}, mary: %{name: "Mary", age: 29,
3 languages: ["Elixir", "F#", "Clojure"]}]
4 [
5   john:%{age:27,languages:["Erlang","Ruby","Elixir"],name:"John"},
6   mary:%{age:29,languages:["Elixir","F#","Clojure"],name:"Mary"}
7 ]
```

Listing 1.21: Struktura koja predstavlja listu korisnika

Prikazana je lista ključnih reči korisnika, gde je svaka vrednost mapa koja sadrži ime, starost i listu programskih jezika koje svaki korisnik voli. Pristup godinama od Džona mogao bi se izvršiti kao na listingu 1.22:

```
1 iex(2)>users[:john].age
2 27
```

Listing 1.22: Pristup godinama od Džona

Ista sintaksa se može koristiti i za ažuriranje vrednosti kao što je dato listingom 1.23:

```
1 iex(3)> users = put_in users[:john].age, 31
2 [
3   john:%{age:31, languages: ["Erlang", "Ruby", "Elixir"], name: "John"},
4   mary:%{age:29, languages: ["Elixir", "F#", "Clojure"], name: "Mary"}
5 ]
```

Listing 1.23: Ažuriranje vrednosti

Makro *update_in/2* je sličan, ali daje mogućnost prosleđivanja funkcije koja kontroliše kako se vrednost menja. Na primer, uklanjanje programskog jezika Clojure sa Marijinog spiska jezika može se uraditi na način prikazan listingom 1.24:

```
1 iex(4)> users = update_in users[:mary].languages,
2 fn languages -> List.delete(languages, "Clojure") end
3 [
4   john:%{age:31, languages: ["Erlang", "Ruby", "Elixir"], name: "John"},
5   mary:%{age:29, languages: ["Elixir", "F#"], name: "Mary"}
6 ]
```

Listing 1.24: Brisanje jezika iz liste

Postoji i funkcija *get_and_update_in* koja omogućava da izvlačenje vrednosti i ažuriranje strukture podataka odjednom. Takođe postoje i funkcije *put_in/3*, *update_in/3* i *get_and_update_in/3* koje omogućavaju dinamički pristup strukturama podataka.

1.4 Osnovni operatori

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

Elixir obezbeđuje 3 bool operatora: **and**, **or** i **not**. Oni su striktni u smislu da očekuju nesto što ima vrednost `true` ili `false` kao svoj prvi operand. Primer koda koji ilustruje ovu osobinu prikazan je na listingu 1.25:

```
1 iex(1) true and true
2 true
3 iex(2) > false or is_atom(:example)
4 true
```

Listing 1.25: Primer upotrebe bool operatora

Ukoliko kao prvi operand prosledimo nesto čija vrednost nije tipa `bool`, dobićemo grešku kao na listingu 1.26:

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

Listing 1.26: Greška pri upotrebi bool operatora

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

Pored ovih boolean operatora, Elixir takođe obezbeđuje operatore `||`, `&&` i `!` koji prihvataju argumente bilo kog tipa. Sve vrednosti osim **false** i **nil** će biti procenjene na `true`, što se može videti na primeru prikazanom listingom 1.27:

```
1 iex(1) > 1 || true
2 1
3 iex(2) > false || 11
4 11
5 iex(3) > nil && 13
6 nil
7 iex(4) > true && 17
8 17
9 iex(5) > !true
10 false
11 iex(6) > !1
12 false
```



```
13 iex(7) > !nil  
14 true
```

Listing 1.27: Operatori koji prihvataju argumente bilo kog tipa

Može se smatrati pravilom da kada se očekuju bool vrednosti, treba koristiti operatore `and` i `or`, a ako bilo koji od operanada ima vrednosti koji nisu tipa bool, onda treba koristiti `||`, `&&` i `!`.

Elixir takođe obezbeđuje `==`, `!=`, `===`, `!==`, `<=`, `>=`, `<` i `>` kao operatore poredjenja, pri čemu se operator `===` od operatora `==` razlikuje po tome što pored vrednosti poredi i tip.

Moguće je i poređenje tipova među sobom. Razlog zbog kojeg se mogu uporediti različiti tipovi podataka je pragmatizam. Algoritmi sortiranja ne moraju da brinu o različitim tipovima podataka da bi sortirali. Ukupan redosled sortiranja je definisan na način prikazan na listingu 1.28:

```
1 number < atom < reference < function < port < pid < tuple  
2 < map < list < bitstring
```

Listing 1.28: Poređenje tipova

1.5 Poklapanje obrazaca

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

Na primer, izraz `2 + 2 = 5` bi rezultirao greškom datom na listingu 1.29:

```
1 iex(1) 5 = 2 + 2
2 ** (MatchError) no match of right hand side value: 4
```

Listing 1.29: Operator uparivanja

Na osnovu greške se može zaključiti da `2 + 2` zaista nije 5. U Elixir-u leva strana mora da ima istu vrednost kao i desna strana. Vrednost izraza dat listingom 1.30 nije greška, već uspešno poklapanje obrazaca:

```
1 iex(1) 4 = 2 + 2
2 4
```

Listing 1.30: Uspešno poklapanje obrazaca

Slično, 2 identična stringa sa obe strane znaka jednakosti će dati rezultat prikazan listingom 1.31:

```
1 iex(1) > "pas" = = "pas"
2 "pas"
```

Listing 1.31: Uspešno poklapanje obrazaca sa stringovima

Poklapanje obrazaca se može prikazati i na pimeru sa listama. Neka je data lista osoba koja je prikazana listingom 1.32:

```
1 iex(1) > lista = ["Milan Stamenkovic", "Petar Jovanovic",
2 "Milica Lazarevic", "Lena Markovic"]
3 ["Milan Stamenkovic", "Petar Jovanovic", "Milica Lazarevic",
4 "Lena Markovic"]
```

Listing 1.32: Lista osoba

Prve tri osobe treba da budu zapamćene, dok četvrta osoba nije bitna. U te svrhe se može iskoristiti poklapanje obrazaca dato na listingu 1.33:

```
1 iex(2) > [prvi, drugi, treci | ostali] = lista
2 ["Milan Stamenkovic", "Petar Jovanovic", "Milica Lazarevic",
```

```
3  "Lena Markovic"]
```

Listing 1.33: Poklapanje obrazaca sa listama

Izvršeno je dodeljivanje prve, druge i treće stavke iz liste promenljivama prvi, drugi i treći. Ostatak liste je dodeljen pomenljivoj ostali pomoću **pipe operatora** (`|`). Vrednost svake od ovih promenljivih može se iščitati na način prikazan na listingu 1.34:

```
1  iex(3)>prvi
2  "Milan Stamenkovic"
3  iex(2)>drugi
4  "Petar Jovanovic"
5  iex(3)>treći
6  "Milica Lazarevic"
7  iex(4)>ostali
8  ["Lena Markovic"]
```

Listing 1.34: Iščitavanje sadržaja promenljivih

1.6 Imutabilnost podataka

U mnogim programskim jezicima je dozvoljeno dodeljivanje vrednosti promenljivoj, a zatim njeno menjanje tokom izvršavanja programa. Mogućnost da zamene vrednost na određenoj memorijskoj lokaciji drugom vrednošću čini se legitimna i čini se da povećava čitljivost našeg programa. Tokom izvršavanja programa obično se ne zna tačno vreme izvršavanje ove promene i obično se i ne vodi računa o tome pri pisanju programe. Ali šta se dešava kada se vrednost u memoriji promeni u trenutku kada je koristi više instanci programa? Neka se ne promeni samo vrednost, već i tip. Ovakvo ponašanje je poznato kao **mutabilnost (promenljivost)**. U konkurentnim okruženjima je izvor grešaka koje je veoma teško pratiti i reprodukovati. Mutabilnost takođe vodi veoma komplikovanom kodu, pisanom ad-hoc kako bi se rešili problemi sinhronizacije. Ovo može smanjiti rizik da konkurentni procesi pristupaju istim resursima, ali po veoma visokoj ceni.

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

1.7 Odlučivanje

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

Elixir obezbeđuje **if/else** uslovne konstrukte kao i mnogi drugi programski jezici. On takođe poseduje naredbu **cond** koja poziva prvu tačnu vrednost koju pronađe. **Case** je još jedan kontrolni tok koji koristi poklapanje obrazaca za kontrolu toka programa.

Elixir ima sledeće vrste naredbi za odlučivanje:

1. *if naredba* - If naredba se sastoji od bool izraza praćenog ključnom reči *do*, jedne ili više izvršnih naredbi i na kraju ključne reči *end*
2. *if..else naredba* - If naredba može biti praćena naredbom *else* (unutar *do..end* bloka), koja se izvršava, ako je bool izraz netačan.
3. *unless naredba* - Naredba *unless* ima isto telo kao i *if* naredba. Kod unutar *unless* naredbe se izvršava samo kada je navedeni uslov netačan.
4. *unless..else* - Naredba *unless...else* ima isto telo kao i naredba *if..else*. Kod unutar *unless..else* naredbe se izvršava samo kada je navedeni uslov netačan.
5. *cond* - Naredba *cond* se koristi ukoliko treba izvršiti neki kod na osnovu nekoliko uslova. Radi kao *if..else if..else* kod drugih programskih jezika.
6. *case* - Naredba *case* se može smatrati zamenom za **switch** naredbu u imperativnim programskim jezicima. Naredba *case* uzima promenljivu ili literal i primenjuje odgovarajući obrazac poklapanja u različitim slučajevima. Ako se bilo koji slučaj poklapa, Elixir izvršava kod povezan sa tim slučajem i izlazi iz *case* naredbe.

1.8 Moduli

U Elixir-u možemo grupisati nekoliko funkcija u module. Već su pomenuti različiti module u prethodnim odeljcima (Map, Enum, List, String,...). Za kreiranje sopstvenih modula u Elixir-u, koristi se makro **defmodule**, a za definisanje svojih funkcija, koristimo makro **def**. Primer koda koji ilustruje kreiranje modula i funkcija dat je listingom 1.35:

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

Listing 1.35: Kreiranje modula i funkcija

Moduli mogu biti ugnježdjeni u Elixir-u. Ova osobina jezika omogućava organizovanje koda na što bolji način. Za ugnježdavanje modula, koristi se sintaksa sa listinga 1.36:

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

Listing 1.36: Ugnježdavanje modula

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

U Elixir-u nema potrebe za definisanjem modula Math, kako bi se definisao modul Math.Adding, pošto jezik prevodi sva imena modula u atome. Mogu se definisati i proizvoljno ugnježdjeni moduli bez definisanja bilo kog modula u lancu. Na primer, može se definisati modul Math.Adding.Sum, iako prethodno nije definisan modul Math i Math.Adding.

1.9 Direktive

Kako bi se olakšala ponovna upotreba koda, Elixir obezbeđuje tri direktive - **alias**, **require** i **import**, kao i makro **use**. Primer njihove upotrebe može se videti na listingu 1.37:

```
1 #Alias modula tako da se mo e pozvati sa Adding umesto sa Math.  
   Adding  
2 alias Math.Adding, as: Adding  
3  
4 #Obezbe uje da je modul kompajliran i dostupan (obi no za makroe)  
5 require Math  
6  
7 #Uklju uje prilago en kod definisan u Math kao pro irenje  
8 use Math
```

Listing 1.37: Primer upotrebe direktiva

Alias

Direktiva **alias** nam služi za podešavanje pseudonima za bilo koje ime modula. Alias mora uvek počinjati velikim slovom. Validni su samo unutar leksičkog opsega u kome su pozvani.

Require

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

Import

Direktivu **import** se koristi kako bi se lakše pristupalo funkcijama i makroima iz drugih modula bez upotrebe potpuno kvalifikovanog imena. **Import** direktiva je takođe leksički određena.

Use

Iako nije direktiva, **use** je makro koji je usko povezan sa zahtevom koji omogućava korišćenje modula u trenutnom kontekstu. Makro use se često koristi od strane programera za unos spoljne funkcionalnosti u trenutni leksički opseg, često modula.

Glava 2

Sekvencioniranje genoma

Rekonstrukcija genoma kroz sekvencioniranje DNK je veoma važan problem u genomici. Ovaj problem izgleda jednostavno, ako se genom može čitati *base-by-base* od 5' ka 3' kraju. Nažalost, postojeće biotehnologije ne mogu proći kroz ceo hromozom, jer je predugačak. Umesto toga, genom se rekonstruiše indirektno. Prvo, vrši se razbijanje genoma na DNA fragmente koristeći pristup očitavanja celog sekvencioniranog genoma (engl. *whole genome shotgun approach*). Zatim se pomoću mašine za sekvencioniranje na osnovu fragmenata vrši dekodiranje DNA sekvence. Ove DNA sekvence se nazivaju **očitanja** (engl. *reads*). Usled slučajnog uzorkovanja, ekstrahirana očitavanja pokrivaju ceo genom ravnomerno. Lepljenjem ovih očitavanja možemo računski rekonstruisati genom. Ovaj proces je poznat kao *de novo* genomska sekvencioniranje.

2.1 Istorija sekvencioniranja genoma

Sanger sekvenciranje je bila najranija tehnika sekvencioniranja. Razvijena su dva asemblera za asembliranje očitavanja Sanger sekvencioniranja: *OLC assembler Celera* i *De Bruijnov graf assembler Ojler*. Čovečiji referentni genom sastavljen je korišćenjem ova dva pristupa. Međutim, kako je Sanger sekvencioniranje niskopropusno i skupo, samo nekoliko genoma je sastavljeno Sanger sekvencioniranjem.

Razvoj sekvencioniranja **druge generacije** promenio je igru. Bilo je moguće efikasno i isplativo sekvencioniranje stotine miliona očitavanja. Međutim, očitavanja druge generacije sekvencioniranja su kratka. Po narudžbini su pravljene genomski asembleri za rekonstrukciju genoma na osnovu kratkih očitavanja. Njihov uspeh je doveo do većeg broja uspešnih **de novo** asemblerskih projekata, uključujući rekonstrukciju genoma **Džejsa Votsona** (engl. *James Watson*) i **panda** genoma. Iako je ovaj pristup ekonomičan, rezultat su bili fragmentisani genomi, jer su očitavanja kratka i regioni koji se ponavljaju dugi.

Nedavno je na raspolaganju sekvencioniranje **treće generacije**, čija su očitavanja duga (dužine od oko 10000 bp). Iako duga očitavanja mogu rešiti redosled ponovljenih regiona, ona imaju visoku stopu greške (15% – 18%). Razvijen je veliki broj računskih metoda za korekciju grešaka u očitavanjima treće generacije sekvencioniranja.

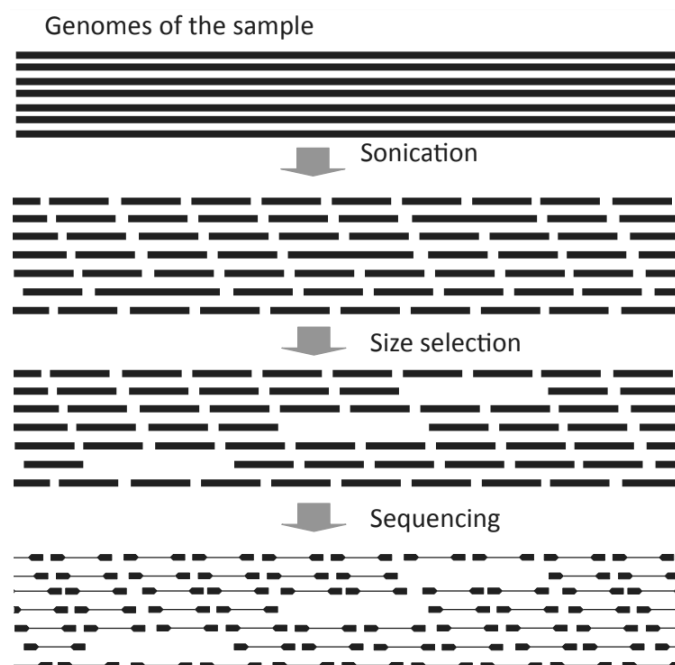
2.2 Čitanje celog sekvencioniranog genoma

Prvi korak u procesu sekvencioniranja genoma je sekvencioniranje skupa očitavanja na osnovu uzorka genoma. Za ovu svrhu postoje 2 protokola:

- sekvencioniranje celog genoma
- sekvencioniranje matičnih parova

Sekvencioniranje celog genoma

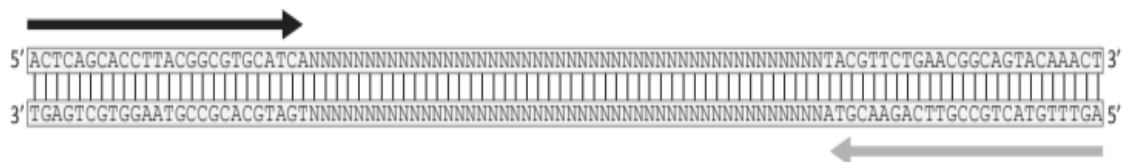
Sekvencioniranje celog genoma uključuje 3 koraka koja su prikazana na slici 2.1:



Slika 2.1: Ilustracija protokola *shotgun* za sekvencioniranje celog genoma. Uključuje 3 koraka: (1) korak razbijanja genoma (*sonication*), (2) korak odabira veličine i izdvajanja fragmenata te veličine (*size selection*), (3) korak čitanja kraj(ev)a svakog fragmenta (*sequencing*)

Prvo, uzorak genoma se na slučajan način razbija na DNA fragmente pomoću *sonication* ili *enzimskog* sečenja. Zatim sledi korak odabira veličine u kom se vrši ekstrahovanje DNA fragmenata određene fiksirane dužine (**veličina umetka**). Na kraju se vrši *single-end* ili *paired-end* sekvencioniranje DNA fragmenata. Za *single-end* sekvencioniranje, sekvencer čita jedan kraj svakog DNA fragmenta, dok za

paired-end sekvencioniranje, sekvencer čita oba kraja DNK fragmenta. (Sekven-
ceri treće generacije čitaju čitav DNA fragment. To se smatra single-end čitanjem.)
Slika 2.2 daje jedan takav primer:



Slika 2.2: Razmatra se DNA fragment ekstrahovan iz uzorka genoma. Pair-end očitavanje se formira izdvajanjem dva DNA podstringa (koji se takođe nazivaju oznake (engl. *tags*)) sa dva kraja DNK fragmenta.

Za single-end sekvencioniranje, dobija se očitavanje sa 5' kraja *forward* šablona DNA fragmenta, tj. ACTCAGCACCTTACGGCGTGCATCA. Za paired-end sekvencioniranje, dobijaju se 5' očitavanja i *forward* i *reverse* šablona DNA fragmenta (u unutrašnjoj orijentaciji, tj. očitavanje šablona od 5' ka 3'), tj:

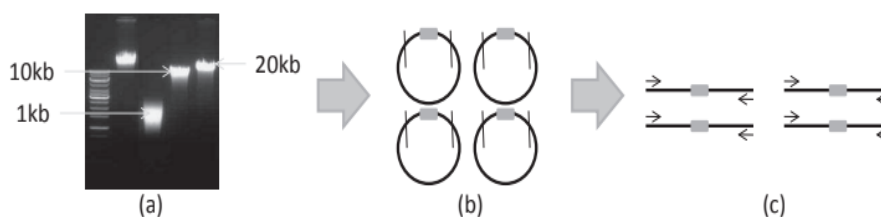
- ACTCAGCACCTTACGGCGTGCATCA
- AGTTTGTACTGCCGTTTCAGAACGTA

Drugo očitavanje je inverzni komplement prvog, jer se ono dobija iz *reverse* šablona.

Za asembliranje genoma, **veličina umetka** (dužina fragmenta DNA) je veoma važan parametar. Različite tehnologije sekvencioniranja imaju različita ograničenja na veličinu umetka. Na primer, *Illumina Hi-seq* sekvencioniranje može izvršiti samo paired-end sekvencioniranje za fragmente DNA veličine umetaka < 1000 bp. Za treću generaciju sekvencioniranja, ograničenje veličine umetka može biti veće od 10000 bp.

Mate-pair sekvenciranje (sekvenciranje očitavanja matičnih parova)

Sekvenceri druge generacije mogu izdvojiti pair-end očitavanja sa oba kraja nekih kratkih fragmenata DNA (veličine umetka manje od 1000 bp). Za izdvajanje pair-end očitavanja dugih fragmenata DNA možemo koristiti **mate-pair sekvenciranje**. Slika 2.3 prikazuje mate-pair proces sekvencioniranja:



Slika 2.3: Ilustracija protokola mate-pair sekvencioniranja očitavanja. (a) Izdvajanje svih fragmenata određene veličine, npr. 10000 bp, gelom za rezanje (cutting gel?). (b) Svaki fragment je cirkularizovan sa adapterom (predstavljen blokom sive boje) u sredini; zatim se vrši sečenje 2 područja oko adaptoru. (c) Konačno, pair-end očitavanja se ekstrahuju pair-end sekvencioniranjem.

Prvo se dugi fragmenti DNA neke fiksirane veličine umetka (npr. 10000 bp) biraju sečenjem gena. Zatim se dugi fragmenti cirkularizuju pomoću adaptoru. Cirkularizovani DNA-ovi su fragmentisani i samo se zadržavaju fragmenti koji sadrže adaptore. Na kraju, pomoću pair-end sekvencioniranja, mate-pair očitavanja se sekvencioniraju na osnovu fragmenata DNA sa adaptorima.

Orijentacija pair-end očitavanja očitnog od strane mate-pair sekvencera se razlikuje od onog koji je očitao od strane pair-end sekvencera. Mate-pair sekvenceri daju dva očitavanja sa oba kraja svakog fragmenta DNA u spoljašnjoj orijentaciji umesto u unutrašnjoj orijentaciji. Npr. za DNA fragment na slici 2.2 mate-pair sekvencioniranje će dati:

- TGATGCACGCCGTAAGGTGCTGAGT
- TACGTTCTGAACGGCAGTACAAACT

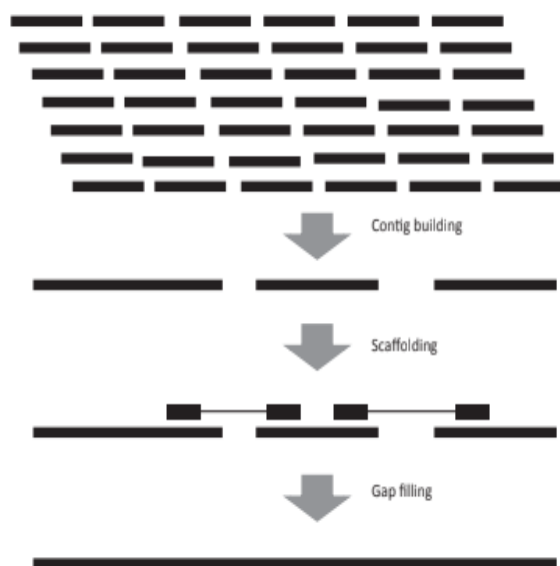
Iako protokol za mate-pair sekvencioniranje može izdvojiti pair-end očitavanja sa velikom veličinom umetka, on zahteva veći broj ulaznih DNA za pripremu sekvencerskih biblioteka i sklon je lažnim greškama.

2.3 De novo sekvencioniranje genoma za kratka očitavanja

Druga generacija sekvencioniranja omogućava dobijanje skupa single-end ili paired-end kratkih očitavanja za ceo genom. De novo sekvencioniranje ima za cilj da izvrši preklapanje očitavanja ispravnom redosledu i rekonstruiše genom.

Problem asembliranja genoma je računski težak. Čak i kada ne postoji greška sekvencioniranja, ovaj problem je ekvivalentan **problemu supstringa** za koji se zna da je NP-kompletan (problem superstringa - na osnovu skupa stringova S , teži se pronalaženju superstringa, najkraćeg stringa P takavog da je svaki string s iz skupa S podstring stringa P). Na primer, ako je $S = \{ACATGC, ATGCGTGT, GTGTACGT\}$, onda je superstring ACATGCGTGTACGT).

Mnogi *de novo* asembleri predlažu asembliranje kratkih očitavanja. Opšte rešenje uključuje 4 koraka prikazana na slici 2.4:



Slika 2.4: Četiri koraka u sekvencioniranju genoma. 1. Korekcija očitavanja: Ispravljanje greškaka sekvencioniranja u očitavanjima. 2. Izgradnja kontiga: Spajanje očitavanja radi formiranja kontigenata. 3. Formiranje skeleta: Korišćenjem paired-end očitavanja se vrši povezivanje kontigenata kako bi se formirali skeleti. 4. Popunjavanje praznina: Za susedne kontige u skeletima, vrši se popunjavanje praznina.

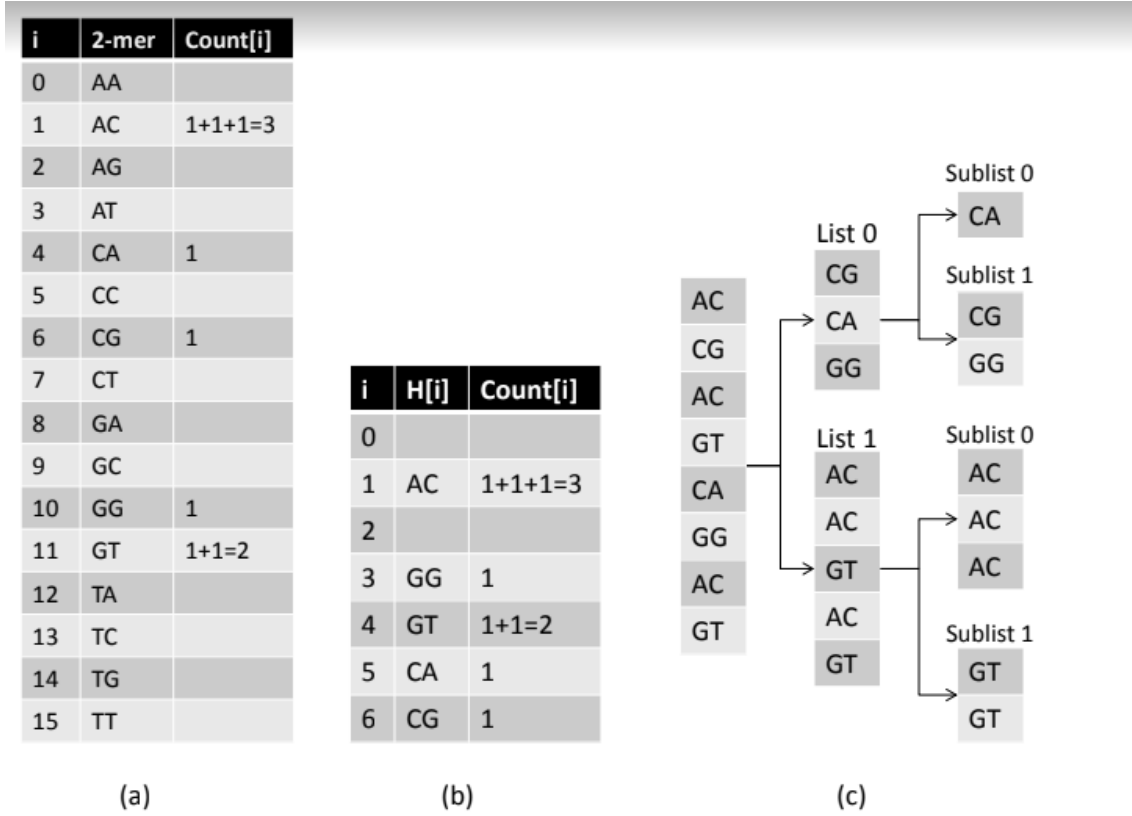
U prvom koraku se ispravljaju greške sekvencioniranja u očitavanjima. Na osnovu korigovanih očitavanja, u drugom koraku se vrši spajanje očitavanja preklapanjem. U idealnom slučaju, želimo da spojimo sva očitavanja tako da formiraju kompletan genom. Zbog ponavljanja se javljaju dvosmislenosti i nije moguće spajanje očitavanja tako da se rekonstruiše kompletan genom. Postojeće metode daju skup kontigenata, pri čemu svaki kontigent predstavlja podregion uzorka genoma. Zatim, koristeći pair-end očitavanja, pokušava se rekonstrukcija redosleda kontigenata tako da se formira skelet. (Svaki skelet je niz kontigenata, a jos se naziva i **superkontig** ili **metakontig**.) Na kraju se vrši preuređivanje očitavanja u skeletima kako bi se popunile praznine između susednih kontigenata.

2.4 k-mer counting

Jedan konceptualno jednostavan, ali osnovni problem je **k-mer brojanje** (k-mer - podstring dužine k). Predstavlja potprogram koji se koristi u korekciji grešaka u očitavanjima. Takođe može biti korišćen u koraku asembliranja, detekciji ponavljanja i kompresiji genomskih podataka. Ulaz je skup očitavanja R i parametar k. Neka je Z skup svih mogućih k-mera koji se pojavljuju u R. Problem je izračunavanje frekvencije pojavljivanja k-mera u Z. U nastavku će biti razmatrana 4 rešenja: (1) **jednostavno heširanje**, (2) **JellyFish**, (3) **BFCOUNTER** i (4) **DSK**.

Jednostavno heširanje

Problem k-mer brojanja može biti rešen implementiranjem asocijativnog niza koristeći heširanje. Kada je k malo (npr. manje od 10), koristi se **savršeno heširanje**. Svaki k-mer z može biti kodiran kao 2k-bitni binarni ceo broj $b(z)$ zamenom A, C, G i T u z sa 00, 01, 10, 11, respektivno. Tako se izgrađuje tabela $Count[0..4^k - 1]$ veličine 4^k u kojoj svaki ulaz $Count[b(z)]$ čuva frekvenciju k-mera z u skupu Z. Preciznije, vrši se inicijalizacija svakog ulaza u $Count[0..4^k - 1]$ na 0. Zatim se iterativno skenira svaki k-mer z iz Z i uvećava $Count[b(z)]$ za 1. Na kraju, svi nenula ulazi u $Count[]$ predstavljaju k-mere koji se pojavljuju u Z kao i broj njihovih pojavljivanja. Slika 2.5(a) predstavlja primer koji ilustruje ovaj jednostavni metod prebrojavanja:



Slika 2.5: Razmatra se skup 4-mera $Z = \{AC; CG; AC; GT; CA; GG; AC; GT\}$: (a) Ilustruje jednostavan metod za prebrojavanje k-mera koji koristi *Count* tabelu veličine $4k$. (b) Ilustruje *JellyFish* metod prebrojavanja k-mera koja koristi heš tabelu veličine 7. Heš funkcija je $h(z) = b(z) \bmod 7$. Na primer, GT se čuva u četvrtom unosu, jer je $h(GT) = 4$. U ovom primeru se javlja jedna kolizija. Pošto je i $h(CA) = 4$, CA je u koliziji sa GT . Linearnim isprobavanjem CA se ipak čuva u petom unosu. (c) Ilustruje DSK metod prebrojavanja k-mera. Pretpostavka je da je $h(z) = b(z)$, $n_{list} = 2$ i $n_{sublist} = 2$. DSK deli Z u 4 ($= n_{list} * n_{sublist}$) podliste, a zatim pokreće *JellyFish* algoritam za prebrojavanje k-mera u svakoj podlisti.

Neka je $N = |Z|$. Tada je navedeni pristup veoma efikasan. Njegovo vreme izvršavanja je $O(N + 4^k)$, a kako treba izgraditi tabelu veličine 4^k , prostorna složenost je $O(4^k)$. Kada je k veliko, navedeni algoritam ne može da radi, jer zahteva previše prostora.

JellyFish algoritam

Moguće je smanjiti veličinu heš tabele koristeći **otvoreni mehanizam adresiranja**. Neka je $h()$ heš funkcija i $H[0..\frac{N}{\alpha} - 1]$ heš tabela koja čuva niz k-mera gde je α faktor opterećenja ($0 < \alpha \leq 1$). Potrebno je izgraditi tabelu $Count[0..\frac{N}{\alpha} - 1]$ gde $Count[i]$ čuva broj za k-mer $H[i]$. Za svaki k-mer iz Z vrši se njegovo heširanje u neki ulaz $H[i]$ gde je $i = h(z)$. Ako $H[i]$ nije prazan i $H[i] \neq z$, ne možemo čuvati z u $H[i]$. Ova situacija se naziva **kolizija**. Kolizija može biti razrešena pomoću otvorenog mehanizma adresiranja. Na primer, kolizija se može razrešiti **linearnim pokušajem**. Ovom metodom pokušavamo da uvećamo indeks i za 1 kada se kolizija dogodi sve dok je $H[i] = z$ ili je ulaz $H[i]$ prazan. Funkcija $hashEntry()$ sa slike 2.6 (donji deo slike) ilustruje šemu linearnog pokušaja za razrešavanje kolizije. Ako $hashEntry(z, h, \frac{N}{\alpha})$ vraća prazan ulaz $H[i]$, onda z ne postoji u heš tabeli i postavljamo $H[i] = z$ i $Count[i] = 1$. U suprotnom, ako $hashEntry(z, h, \frac{N}{\alpha})$ vraća ulaz $H[i] = z$, uvećavamo $Count[i]$ za jedan. Nakon sto su svi k-meri iz Z obrađeni, prikazujemo $(H[i], Count[i])$ za sve nenula ulaze $H[i]$.

Algorithm Jellyfish(Z, α, h)

Require: Z is a set of N 's k -mers, α is a load factor that controls the hash table size and $h(.)$ is the hash function

Ensure: The count of every k -mer appearing in Z

- 1: Set $H[1..\frac{N}{\alpha}]$ be a table where each entry requires $2k$ bits
- 2: Set $Count[1..\frac{N}{\alpha}]$ be a table where each entry requires 32 bits
- 3: Initialize T to be an empty table
- 4: **for** each k -mer z in Z **do**
- 5: $i = hashEntry(z, h, \frac{N}{\alpha});$
- 6: **if** $H[i]$ is empty **then**
- 7: $H[i] = z$ and $Count[i] = 1;$
- 8: **else**
- 9: $Count[i] = Count[i] + 1;$
- 10: **end if**
- 11: **end for**
- 12: Output $(H[i], Count[i])$ for all non-empty entries $H[i];$

Algorithm hashEntry($z, h, size$)

- 1: $i = h(z) \bmod size;$
- 2: **while** $H[i] \neq z$ **do**
- 3: $i = i + 1 \bmod size;$ /* linear probing */
- 4: **end while**
- 5: Return $i;$

Slika 2.6: Jellyfish algoritam i funkcija hashEntry

JellyFish algoritam je detaljnije objašnjen na slici 2.6 (gornji deo slike), dok slika 2.5 (b) daje primer koji ga ilustruje. On je efikasniji, ukoliko ne postoji kolizija. U praksi je očekivani broj kolizija manji, ukoliko za faktor opterećenja važi $\alpha \leq 0.7$. Zatim, očekivano vreme izvršavanja je $O(N)$. Što se tiče prostorne složenosti, tabele $H[]$ i $Count[]$ zahtevaju $\frac{N}{\alpha}(2k + 32)$ bitova, pod pretpostavkom da broj zauzima 32 bita. Gore pomenuta ideja smanjivanja veličine heš tabele je iskorišćena u JellyFish algoritmu.

Iako *JellyFish* algoritam koristi manje prostora od metode naivnog prebrojavanja, *JellyFish* heš tabela mora biti veličine koja je jednaka bar broju jedinstvenih k-mera iz Z . *JellyFish* i dalje zahteva mnogo memorije u slučaju da je broj jedinstvenih k-mera u Z veliki.

BFCOUNTER algoritam

U mnogim aplikacijama, od značaja su samo k-meri koji se pojavljuju najmanje q puta. Kada bi moglo da se izbegne čuvanje k-mera koji se pojavljuju manje od q puta, sačuvalo bi se mnogo memorije. Pol Melsted (engl. *Páll Melsted*) je predložio algoritam **BFCOUNTER** koji prebrojava samo k-mere koji se pojavljuju najmanje q puta. On koristi *counting Bloom* filter da odredi da li se k-mer pojavljuje najmanje q puta. To je prostorno efikasna probabilistička struktura podataka koja dozvoljava dodavanje bilo kojih k-mera u nju i ispitivanje da li se k-mer pojavljuje najmanje q puta. Iako on može dati pogrešno pozitivan rezultat (pogrešan izveštaj da k-mer postoji ili pogrešno proceniti broj k-mera), ne može dati pogrešno negativan rezultat. BFCOUNTER održava *counting Bloom* filter B i heš tabelu H . Sastoji se od 2 faze. Prva faza počinje sa praznim *counting Bloom* filterom B i praznom heš tabelom H . U ovoj fazi se skeniraju k-meri iz Z jedan po jedan. Za svaki k-mer z se proverava da li se z pojavljuje najmanje $q - 1$ puta u *counting Bloom* filteru B utvrđivanjem da li je $countBloom(x, B) \geq q - 1$. Ako nije, vrši se umetanje z u Z pomoću $insertBloom(z, B)$. U suprotnom, z se pojavljuje najmanje q puta. Zatim se vrši proveravanje da li je z u heš tabeli H . Ako nije, z se umeće u neki prazan ulaz $H[i]$ i postavlja se $count[i]$ na nulu.

U drugoj fazi se obavlja stvarno brojanje. Vrši se skeniranje k-mera iz Z jedan po jedan. Za svaki k-mer z iz Z , ako se z pojavljuje u heširanom ulazu $H[i]$, onda se uvećava $count[i]$ za 1. Detaljan pseudokod je prikazan na slici 2.7:

Algorithm BFCOUNTER(Z, q, α, h)

Require: Z is a set of N 's k -mers, α is a load factor that controls the hash table size and $h(\cdot)$ is the hash function

Ensure: For every k -mer in Z occurring at least twice, report its count.

- 1: Set $H[1..\frac{N}{2\alpha}]$ be an empty hash table where each entry requires $2k$ bits;
- 2: Set $Count[1..\frac{N}{q\alpha}]$ be an empty vector where each entry requires 32 bits;
- 3: Set $B[0..m-1]$ be an empty vector where each entry requires $\lceil \log q \rceil$ bits and $m = 8N$;
- 4: **for** each k -mer z in Z **do**
- 5: **if** $\text{countBloom}(z, B) \geq q - 1$ **then**
- 6: $i = \text{hashEntry}(z, h, \frac{N}{q\alpha})$;
- 7: **if** $H[i]$ is empty **then**
- 8: Set $H[i] = z$ and $Count[i] = 0$;
- 9: **end if**
- 10: **else**
- 11: $\text{insertBloom}(z, B)$;
- 12: **end if**
- 13: **end for**
- 14: **for** each k -mer z in Z **do**
- 15: $i = \text{hashEntry}(z, h, \frac{N}{q\alpha})$;
- 16: **if** $H[i] = z$ **then**
- 17: $Count[i] = Count[i] + 1$;
- 18: **end if**
- 19: **end for**
- 20: **Output** ($H[i], Count[i]$) for all non-empty entries $H[i]$ where $Count[i] \geq q$;

0

Slika 2.7: Prostorno efikasan algoritam za prebrojavanje k -mera koji broji samo k -mere koji se pojavljuju najmanje q puta.

Vreme izvršavanja BFCOUNTER algoritma je $O(n)$. Što se tiče prostorne složenosti, *counting Bloom* filter zahteva $O(N \log(q))$. Prostor za $H[]$ i $Count[]$ je $\frac{N'}{\alpha}(2k + 32)$ bitova, gde je N' broj k -mera koji se pojavljuju najmanje q puta. Primetimo da je $N' \leq \frac{N}{q}$.

DSK algoritam

Iako je BFCounter prostorno efikasan, njegova prostorna složenost i dalje zavisi od broja N k-mera u Z . Neka je memorija fiksirana tako da bude M bitova i neka je memorija diska fiksirana tako da bude D bitova. Da li se može i dalje efikasno izračunati pojavljivanja k-mera? Gijom Rizk (engl. *Guillaume Rizk*) nam daje pozitivan odgovor i predlaže metod koji se naziva **DSK**. Ideja ovog metoda je da se skup k-mera Z podeli u različite liste tako da svaka lista bude smestena na disk koristeći D bitova. Zatim, za svaku listu, k-meri iz liste se dalje dele u podliste tako da svaka podlista može biti sačuvana u memoriji koristeći M bitova. Na kraju, frekvencije k-mera u svakoj podlisti se izračunavaju algoritmom *JellyFish* sa slike 2.6.

Preciznije, k-meri u Z su podeljeni u n_{list} lista približno slične dužine. Kako disk ima D bitova i svaki k-mer može biti reprezentovan u $2k$ bitova, svaka lista može čuvati $l_{list} = \frac{D}{2k}$ k-mera. Kako imamo N k-mera u Z , postavlja se $n_{list} = \frac{N}{n_{list}} = \frac{2kN}{D}$. Ovo deljenje se obavlja heš funkcijom $h()$ koja ravnomerno mapira sve k-meru u n_{list} lista. Preciznije, za svaki k-mer z iz Z , z se dodeljuje i -toj listi, ako je $h(z) \bmod n_{list} = i$.

Zatim, svaka lista se dalje deli u podliste, pri čemu je svaka dužine $l_{sublist}$. Svaka podlista će biti obrađena u memoriji pomoću algoritma *JellyFish*, koji zahteva $\frac{l_{sublist}}{0.7}(2k + 32)$ bitova. Kako memorija ima M bitova, tako je $l_{sublist} = \frac{0.7M}{(2k+32)}$.

Broj podlista je jednak $n_{sublist} = \frac{n_{list}}{n_{sublist}} = \frac{D(2k+32)}{0.7(2k)M}$. Slično, svaka lista je podeljena u podliste heš funkcijom $h()$. Preciznije, za svaki k-mer s u i -toj listi, s je dodeljeno j -toj podlisti, ako je $(\frac{h(s)}{n_{list}}) \bmod$

$$n_{sublist} = j.$$

Za svaku podlistu dužine $l_{sublist} = 0.7M2k + 32$, koristeći M bitova, brojimo broj pojavljivanja svakog k-mera u podlisti koristeći *JellyFish*($d_j, 0.7, h$) sa slike 2.6.

Na slici 2.5(c) se može videti primer koji ilustruje izvršavanje algoritma DSK. Neka je $n_{list} = 2$, $n_{sublist} = 2$ and $h(z) = b(z)$ za svaki $z \in Z$. Kako je $n_{list} = 2$, algoritam izvršava 2 iteracije (u nastavku sledi opis nulte iteracije, jer se prva izvršava slično). Prva faza nulte iteracije skenira sve k-meru iz Z i identifikuje svaki k-mer $z \in Z$ koji pripada nultoj listi. Na primer, $h(GG) = 10$, kako je $h(GG) \bmod n_{list} = 0$ i $\frac{h(z)}{n_{list}} \bmod n_{sublist} = 1$, GG pripada nultoj listi i prvoj podlisti. Nakon toga, nulta lista se deli na nultu podlistu $\{CA\}$ i prvu podlistu $\{CG, GG\}$. Obe podliste su zapisane na disku. Druga faza čita svaku podlistu iz memorije i broji k-meru koristeći *JellyFish* algoritam.

Ovaj algoritam će zapisati samo jednom svaki k-mer iz Z, iako će svaki k-mer pročitati n_{list} puta. Stoga, on neće generisati mnogo pristupa disku radi pisanja. Što se tiče vremenske složenosti, za i-tu iteraciju, algoritam numeriče sve k-mere u Z, što oduzima $O(n)$ vremena. Zatim, algoritam identifikuje $\frac{D'}{2k}$ s k-mera koji pripadaju i-toj listi i zapisuje ih na disk, što oduzima $O(\frac{D}{k})$ vremena. Nakon toga, algoritam čita $\frac{D'}{2k}$ s k-mera i izvodi brojanje, što oduzima $O(\frac{D}{k})$ vremena. Tako da svaka iteracija zahteva $O(N + \frac{D}{k}) = O(N)$ vremena, gde je $N > \frac{D}{2k}$. Kako je $n_{list} = \frac{2kN}{D}$ broj iteracija, algoritam se izvršava u $O(kN^2)$ očekivanom vremenu. Kad je $D = \theta(N)$, algoritam se izvršava u $O(kN)$ očekivanom vremenu.

Glava 3

Zaključak

Biografija autora

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