Programski jezici svojstveni za problemsku domenu

Smoljan, Antonio

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Antonio Smoljan

PROGRAMSKI JEZICI SVOJSTVENI ZA PROBLEMSKU DOMENU

ZAVRŠNI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Antonio Smoljan

Matični broj: 0016123808

Studij: Informacijski sustavi

PROGRAMSKI JEZICI SVOJSTVENI ZA PROBLEMSKU DOMENU

ZAVRŠNI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, kolovoz 2019.

Antonio Smoljan

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovome radu će se obraditi tema programskih jezika svojstvenih za problemsku domenu. U početku se definira to što su jezici svojstveni za problemsku domenu te se navede podjela i par primjera. Kao primjer programskog jezika svojstvenog za problemsku domenu, u ovome radu se opisuje sintaksa i struktura programskog jezika R u kojemu se realizira jednostavni projekt. Projekt uključuje analiziranje seta podataka sa šahovskim partijama te se kroz par primjera prikazuje u kakvoj su ovisnosti pobjednik partije, broj poteza, način otvaranja i druge stvari.

Ključne riječi: Programski jezici svojstveni za problemsku domenu; Statističko programiranje; Programski jezik R; RStudio; Analiza seta podataka; Analiza šahovskih partija.

Sadržaj

1. Uvod

Često ponavljanje nekih problema u nekom programskom jeziku zahtjeva pojednostavljenje rješavanja tog problema. U tu svrhu se razvija velik broj jezika koji služe samo za taj problem i njihov opseg djelovanja je ograničen samo na taj problem ili možda još nekoliko drugih problema.

Jezici namijenjeni za opću domenu (engl. General-purpose language) kao što su Python, Java, C++ i drugi su namijenjeni za široku primjenu te se koriste unutar raznih domena. Međutim, za neke specifične domene može postojati jezik koji značajno olakšava posao developeru te nudi bolje rješenje. Takav jezik pripada skupini jezika svojstvenih za problemsku domenu.

U ovome radu će se obrađivati upravo ti programski jezici svojstveni za problemsku domenu te će se objasniti njihove podjele i prednosti i mane.

Jedan od primjera programskih jezika svojstvenih za problemsku domenu je programski jezik R čija je sintaksa i struktura podataka objašnjeni u ovome radu. Tako je pokazan način na koji funkcionira konstrukt *if()*, petlja *for()*, kako rade funkcije, pokazane su strukture podataka poput vektora, višedimenzionalnih vektora i lista. Također, pokazano je kako se kreiraju nacrti i grafikoni.

U svrhu rada je napravljen jednostavni projekt u kojemu je, u programsku jeziku R, analiziran set podataka koji sadrži informacije o šahovskim partijama. Šah je igra koja je se dosta analizirala zbog svoje jednostavnosti. Analiza uključuje nekoliko primjera u koje su uključeni grafikoni te pokazuje ovisnosti pobjednika partije, boje igrača, broja poteza na partiji, načina otvaranja.

2. Jezici svojstveni za problemsku domenu

2.1. Uvod

Sami pojam jezika svojstvenih za problemsku dilemu je malo nejasan jer nije definirano gdje staje jezik namijenjen za opću domeni i počinje jezik svojstven za problemsku domenu. Kao što sami naziv kaže, jezici svojstveni za problemsku domenu (engl. Domain-Specific Languages) su namijenjeni za usku primjenu (Mernik, Heering, & Sloane, 2011) i kao takvi su suprotnost jezicima svojstvenim za opću domenu. Time što su namijenjeni za specifičnu domenu te kao takvi su više orijentirani na programera što zna pomoći programerima u različitim situacijama. Samim time, jezici četvrte generacije su obično DSL-ovi. (Mernik et al., 2011) Činjenica da su programski jezici četvrte generacije i DSL-ovi usko vezani pomaže u samom shvaćanju DSL-ova. Možemo reći da, u evoluciji programskih jezika, DSL-ovi su korak iznad GPL-ova.

2.2. Primjeri DSL-ova

Kao primjer možemo navesti SQL kojega dosta stručnjaka svrstava u DLS-ove. SQL je definitivno jedan od najkorištenijih DSL-ova i jedan od najkorištenijih jezika uopće. Služi za upravljanje bazama te je praktički bez konkurencije kad je upravljanje bazama u pitanju.

Martin Fowler u svojoj knjizi Domain-Specific Languages (Martin Fowler, 2010) kao primjer DSL-a navodi regularne izraze. Za primjer uzmimo regularni izraz za unos E-maila[1] koji je prikazan ispod.

```
/^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/
```

Iako regularni izrazi nisu laki za shvatiti, pokušajmo zamisliti koliko bi linija koda zauzelo da smo istu funkciju napisali čistim kodom u nekom od gore navedenih GPL-ova i koliko bi taj kod bio čitljiv i jednostavan za izmijeniti. Dakle, DSL olakšava developeru utoliko da se kod može brže napisati, brže i jednostavnije za izmijeniti te su manje šanse za pojavljivanje grešaka u kodu (Martin Fowler, 2010).

Nakon ovoga definiranja DSL-ova, nije teško zaključiti kako se njihovo korištenje uglavnom preporučuje, ali se treba shvatiti da se sa njima može napraviti ograničen spektar stvari te se ne treba očekivati ništa više od onoga za što su namijenjeni. Tipični projekt može koristiti desetak takvih jezika (Martin Fowler, 2010).

[1] izvor: https://code.tutsplus.com/tutorials/8-regular-expressions-you-should-know--net-6149

2.3. Podjele DSL-ova

Postoji velika raznolikost DSL-ova te tako ih možemo podijeliti na više načina. Jedna je podjela na interne i eksterne DSL-ove, a druga na Fragmentarni i samostalni oblik DSL-a.

2.3.1. Interni i eksterni DSL

Interni DSL je jezik razvijen unutar određenog domaćinskog jezika. Taj jezik koristi sintaksu i pravila domaćinskog jezika, ali je dovoljno različit da se smatra prilagođenim jezikom. Interni jezici se mogu pisati u svakome GPL jeziku, ali u zadnje vrijeme se Ruby smatra predvodnikom ovog pristupa. Jedan od najpopularnijih frameworkova Ruby-a, Ruby on Rails smatra se skupom DSL-ova (Martin Fowler, 2010).

Za razliku od internog DSL-a, eksterni DSL posjeduje vlastita pravila i vlastitu sintaksu te ne ovisi o domaćinskom jeziku (što ne znači da ne može bit ista sintaksa kao na domaćinskom jeziku). Ovi DSL-ovi su lakši za primijetiti od internih jezika te velik dio poznatih DSL-ova pripada upravo ovoj vrsti. (Martin Fowler, 2010) Već navedena dva primjera, SQL i regularni izrazi pripadaju eksternim DSL-ovima.

2.3.2. Fragmentarni i samostalni oblik DSL-a

DLS-ovi mogu dolaziti u dva oblika. To su fragmentarni i samostalni oblik (Martin Fowler, 2010).

Fragmentarni oblik podrazumijeva da se DLS kod može miješati sa kodom programa pa taj oblik možemo promatrati kao proširenje GPL-a. Primjer Fragmentarnog oblika su već spomenuti regularni izrazi (Martin Fowler, 2010).

Kod samostalnog oblika, DLS dio koda dolazi u zasebnoj datoteci koja se poziva u glavnom programu (Martin Fowler, 2010). Većina takvih jezika ima zasebno okruženje u kojem se piše kod. Kasnije u ovome radu će se obrađivati R programski jezik te on pripada skupini jezika samostalnog oblika.

Spomenuti SQL je iznimka tako što može dolaziti u oba oblika.

2.4. Prednosti i mane DSL-ova

Implementiranje DSL-a u određeni projekt donosi neke prednosti i mane. Dobro dizajniran DSL bi trebao nuditi dobar balans prednosti (van Deursen, Klint, & Visser, 2000). Također, važna je uloga menadžera koji odlučuje hoće li se implementirati DSL jer, unatoč

tome što taj DSL može bit dobro dizajniran sam po sebi, jednostavno se ne isplati radi raznih mana koje su navedene ispod.

Pravilno korišteni DSL-ovi mogu povećati produktivnost, pouzdanost, održivost i prenosivost. Mogu biti ponovo korišteni za druge svrhe, mogu povećati mogućnost testiranja (van Deursen et al., 2000).

Unatoč mnogim prednostima, DSL-ovi se ne smiju uvoditi bezuvjetno. Neki od tih uvjeta je trošak izrade (ili licenciranja), implementiranja i održavanja DSL-a, također tu je trošak edukacije korisnika DSL-a. Također, potrebno je znati jasno definirati domenu DSL-a i kad ga koristiti, a kad ne. I tu je bitno da programer bude iskusan (van Deursen et al., 2000).

3. Uvod u analizu šaha

Kako bi jasnije objasnili programiranje u jeziku R, u ovome radu će se analizirati jedan set podataka koji prikazuje preko 20000 šahovskih partija na stranici *lichess.org*.

Šah je igra na ploči sa velikom povijesti, nije potrebno opisivati igru jer je svima poznata, ali je potrebno naglasiti da je dugi niz godina predmet analize statističara, „Big data" znanstvenika i mnogih drugih analitičara. Već 1997. godine, računalo *Deep Blue* je, uz pomoć umjetne inteligencije, porazilo tadašnjeg svjetskog prvaka u šahu, Garija Kasparova (IBM, 2012).

Set podataka je objavljen na stranici Kaggle 2017. godine od strane korisnika pod korisničkim imenom Mitchell J[2]. Mali problem ovoga seta podataka je taj da nisu objašnjena značenja svih kolona pa neke kolone neće biti jasne za sve.

Set podataka dolazi u jednoj *.csv* datoteci što olakšava prilagodbu za jezik R (nema potrebe spajati više datoteka). Sama datoteka je teška oko 7.5 MB. Set podataka u sebi sadrži informacije sa 20058 šahovskih partija koje su odigrane na stranici *lichess.org* te ne nedostaju nikakve informacije. Podatci su raspoređeni u 16 kolona (14 se koristi u ovome radu jer su dvije nepotrebne).

Set podataka o svakome meču sadrži sljedeće podatke:

- *id* – ID meča (ne koristi se)
- *rated* – ulazi li partija u bodovanje (ne koristi se)
- *created_at* – kad je partija počela
- *last_move_at* – kad je partija završila (ne koristi se)
- *turns* – broj poteza u partiji
- *victory_status* – način pobjede
- *winner* – pobjednik (crni ili bijeli)
- *increment_code* – prikazuje koliko partija traje i koja je inkrementacija nakon svakog poteza (ne koristi se)
- *white_id* – ID bijelog igrača
- *white_rating* – ocjena bijelog igrača
- *black_id* – ID crnog igrača
- *black_rating* – ocjena crnog igrača

[2] link s kojeg je set podataka preuzet: https://www.kaggle.com/datasnaek/chess

- *moves* – svaki potez u partiji (ne koristi se)
- *opening_eco* – ECO kodovi za otvaranje
- *opening_name* – naziv otvaranja (ne koristi se)
- *opening_ply* – broj poteza tokom otvaranja (ne koristi se)

Cilj analiziranja ovog seta podataka je uvid u čimbenike koji utječu na ishod meča. Npr. koje otvaranje je najučinkovitije i na koji način se pobjeđuje, je li bolje biti bijeli ili crni igrač itd.

4. Programski jezik R

4.1. Statističko programiranje

Nije lako definirati što je statističko programiranje ponajviše zato što statističari mogu raditi širok raspon stvari koje će raditi i obični programer. Statističari su zaduženi za skupljanje i analiziranje podataka. Statističko programiranje uključuje izračune koji pomažu u statističkoj analizi, tj. sumiranje podataka i prikazivanje rezultata. Bitan dio statističke analize je crtanje grafova i stohastičke simulacije pa je bitno da jezik u kojemu se programira podržava oboje (Braun & Murdoch, 2007). Pored programskog jezika R, među poznatijim primjerima programa koji služe za statistiku spada Microsoft Excel, SAS, SPSS, S-plus i drugi.

4.2. Povijest

R je programski jezik i besplatna razvojna okolina namijenjena za statističko računanje i izradu grafova (r-project.org, 2019). Razvijen je po uzoru na statistički jezik S i najkompatibilniji je s njim (Matloff, 2009) što znači da većina koda napisanog u jeziku S će raditi i u jeziku R (r-project.org, 2019). R je razvijen 1993. godine od strane Rossa Ihake i Roberta Gentlemana na sveučilištu u Aucklandu (Novi Zeland) kao implementacija na jezik S (Braun & Murdoch, 2007) i trenutno ga razvija *R Development Core Team*. U trenutku pisanja ovoga rada (kolovoz 2018.), najnovija verzija je 3.6.1 koja je izašla u srpnju 2019. godine čime možemo zaključiti da je R jedan jako živ jezik i okruženje. Tome u prilog govori i podatak da godišnje prosječno izlazi 5 stabilnih verzija. Dostupan je na Windowsima, Macu i Linuxu.

Danas, R je open source¸ što znači da je besplatan i da se lako može vidjeti kako je napisan i može se unaprijediti (Braun & Murdoch, 2007). S vremenom je postao standard među profesionalnim statističarima (Matloff, 2009).

R je skriptni jezik što znači da se kod može izvršavati na gotov program.

4.3. RStudio

Okruženje koje se koristi u ovom radu za razvijanje u R-u je RStudio.

RStudio je integrirano razvojno okruženje za R. Uključuje konzolu, editor koji podržava direktno izvršavanje koda, kao i alate za crtanje grafova, otklanjanje grešaka i upravljanje radnim prostorom (RStudio, 2019).

Dolazi u verzijama za desktop i za server.

RStudio je dostupan u open source varijanti i varijanti za komercijalnu upotrebu. Može raditi kao desktop aplikacija i preko Internet preglednika tako da je spojen na RStudio Server verziju (RStudio, 2019).

4.4. Instalacija

Sama instalacija programskog jezika R i njegovog okruženja je jednostavna.

Prvo što se treba uraditi je instalirati R-project. To se može učiniti na ovoj poveznici: https://cran.r-project.org/bin/windows/base/. Na toj poveznici, osim poveznice za preuzimanju najnovije verzije R-projecta, može se naći poveznica za detaljnije instrukcije o preuzimanju, informacije o novostima u najnovijoj verziji te se mogu naći prijašnje verzije. Klikom na poveznicu za preuzimanje najnovije verzije, preuzima se datoteka koja služi za instalaciju R-projecta.

Instalacija je jednostavna i ne bi trebalo biti problema s tim. Nakon odabiranja jezika instalacije, lokacije gdje se želi instalirati R-project i specificiranja komponenata koje nam točno trebaju, program se instalira.

Pošto je R-project konzolna aplikacija, da bi imali okruženje potreban nam je RStudio. Na sljedećoj poveznici se može preuzeti RStudio: https://www.rstudio.com/. Ovisno o tome što nam treba, tu možemo preuzeti besplatne verzije za desktop i za server te možemo kupiti komercijalne licence za desktop i za server. Komercijalna licenca za desktop košta 995 američkih dolara godišnje, a za server 4,975 američkih dolara s tim da istu licencu može koristiti do pet korisnika. Nakon što preuzmemo datoteku za RStudio, kao i kod R-projecta, slijedi jednostavna instalacija u kojoj se bira lokacija gdje se želi instalirati RStudio.

4.5. Interaktivni i *batch* modovi

R ima dva moda, interaktivni i batch (Matloff, 2009).

U interaktivnom modu se mogu izvršiti R naredbe (Matloff, 2009). U interaktivnom modu se svaka naredba izvršava zasebno.

U *batch* modu se može pokrenuti više R naredbi te se proces izvođenja automatizira (Matloff, 2009).

4.5.1. Pokretanje u *batch* modu

Moguće je pokrenuti R skriptu spremljenu u *.R* datoteci iz drugih programa.

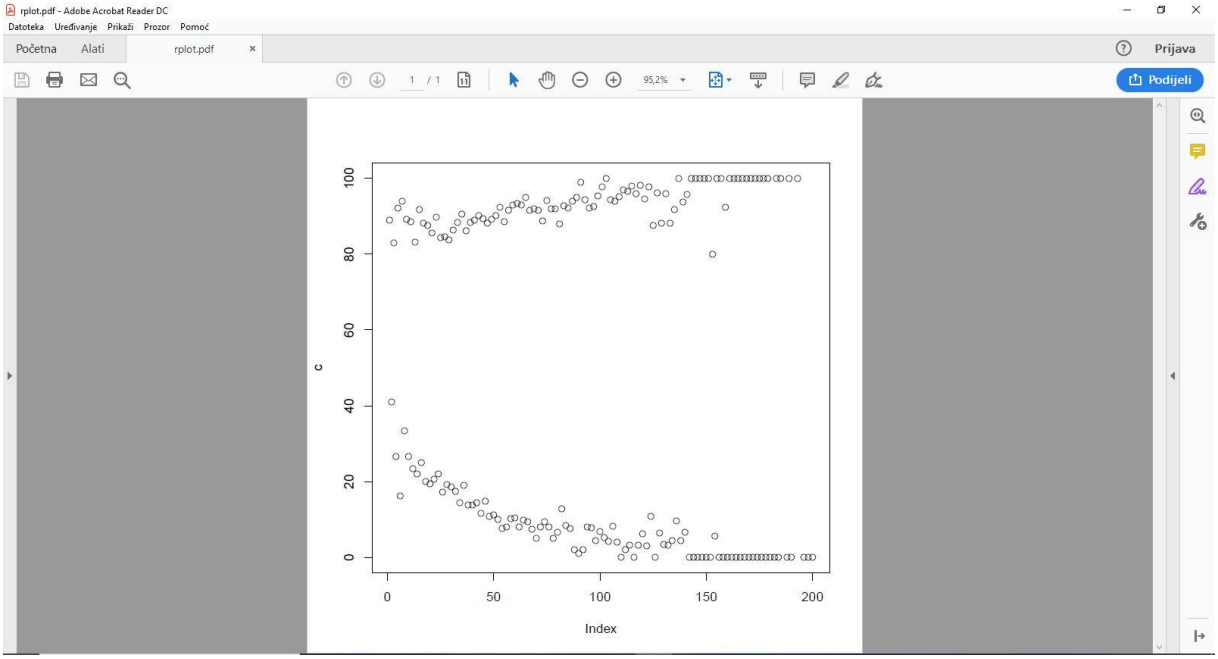
Da bismo to pokazali imamo jednostavnu skriptu koja kreira graf u *.pdf* formatu. Sadržaj *.R* datoteke je sljedeći:

```
sah<-read.csv(file="C:\\sah\\games.csv", header=TRUE)
a<-NULL
b<-NULL
for(i in 1:200){
  a=c(a,0)
  b=c(b,0)
}
for(i in 1:20058){
  if(sah$turns[i]<=200 &sah$turns[i]>=1){
    if(sah$winner[i]=="black"){
      b[sah$turns[i]]=b[sah$turns[i]]+1
    }else if(sah$winner[i]=="white"){
      a[sah$turns[i]]=a[sah$turns[i]]+1
    }
  }
}
c<-NULL
for(i in 1:200){
  c=c(c,100*a[i]/(a[i]+b[i]))
}
pdf("C:\\sah\\rplot.pdf")
plot(c)
dev.off
```

Zasad je bitno samo da ova skripta kreira *.pdf* datoteku na sljedećoj putanji: *C:\sah\rplot.pdf*.

Ova skripta se može pokrenuti iz *cmd*-a pomoću sljedeće naredbe: *R* (ili putanja do *R.exe*) *CMD BATCH* [putanja do *.R* datoteke].

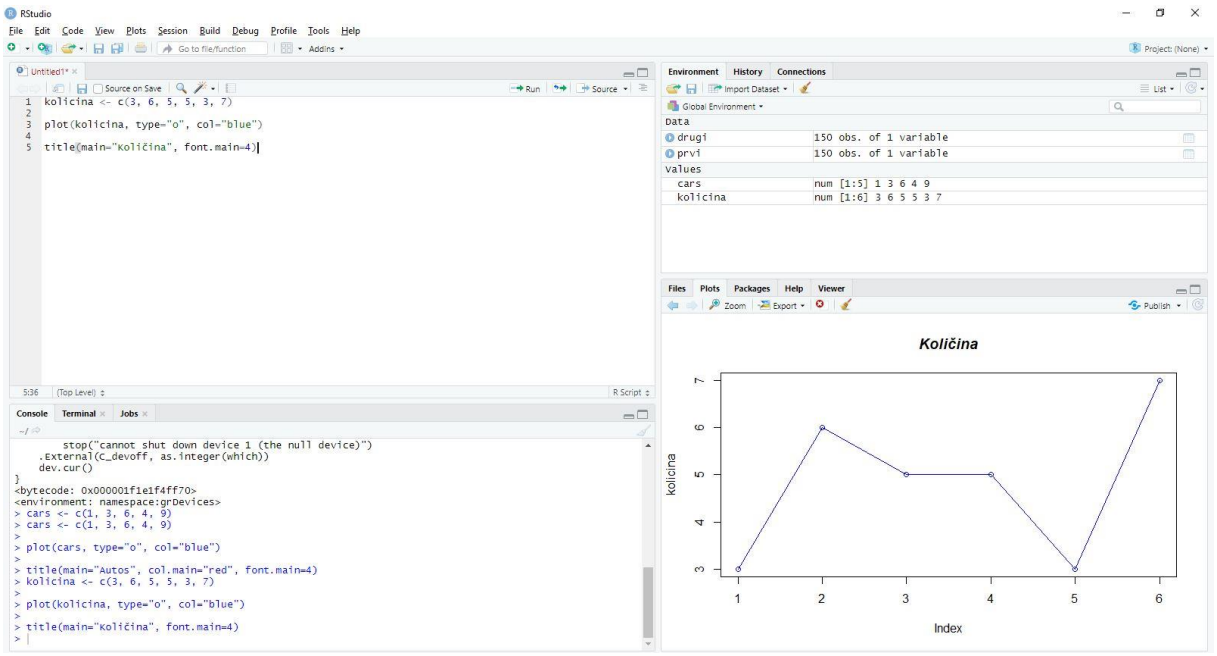
Nakon izvršavanja skripte, na prije spomenutoj putanji se pojavi *.pdf* datoteka sa grafom (slika 1.).



Slika 1. PDF datoteka koja je dobivena izvršavanjem R skripte

4.6. Sučelje RStudia

Zadano sučelje RStudia je lako shvatljivo te, kao što možemo vidjeti na slici 1., podijeljen je u četiri dijela.



Slika 2. Sučelje RStudia

U gornjem lijevom kutu se nalazi sami uređivač koda preko kojega se unosi kod i pokreće isti. Kako je R skriptni jezik, kod se izvršava tako što se odabere dio koda koji se želi izvršiti te se klikne gumb *Run*.

U donjem lijevom kutu se nalazi konzola gdje vidimo izvršeni kod i terminal. (Na slici je otvorena konzola)

U gornjem desnom kutu se nalaze kartica *Environment* gdje možemo vidjeti već zauzetu memoriju u vidu varijabla, kartica *History* gdje vidimo do sad izvršeni kod te karticu *Connections* gdje vidimo sve otvorene veze. (Na slici je otvoren *Environment*)

U donjem desnom kutu se nalaze kartica *Files* gdje možemo pregledavati datoteke, kartica *Plot* gdje vidimo skicu (graf) koja je napravljena, karticu *Packages* gdje možemo pregledavate određene pakete i dodavati iste u projekt i karticu *Help* koja služi za pomoć.

4.7. Komentari

Kao i svi ostali jezici, unutar R jezika se mogu pisati komentari koje će interpreter zanemariti. Postoje samo jednolinijski komentari (višelinijski ne postoje) i da bi smo pisali takve komentare, koristimo znak „#“ (Matloff, 2009). Na dijelu koda ispod možemo vidjeti jasnije kako funkcioniraju komentari u jeziku R.

```
ovo nije komentar
ovo nije komentar  #ovo jest komentar
ovo nije komentar
#ovo jest komentar
ovo nije komentar
```

Komentari ne mogu preći u novi red te u istom redu možemo imati i dio koda i komentar s tim da prvo mora doći dio koda. Prilikom izvršavanja koda, R, osim što sačuva kod koji je se izvršio, sačuva i komentare unutar *History* kartice.

4.8. Dobivanje pomoći

R ima nekoliko ugrađenih funkcija koje su dizajnirane za dobivanje potrebne pomoći.

Jedna takva je funkcija *help()* koja kao argument prima ime druge funkcije za koju nam trebaju dodatna objašnjenja (Braun & Murdoch, 2007). Npr. ako želimo dobiti pomoć za funkciju *mean()* onda ćemo u argument funkcije *help()* napisati *mean* te će to izgledati ovako: *help(mean).* Kad se ta funkcija pokrene, onda se u *help* kartici (donji desni dio sučelja) otvori stranica sa službenom dokumentacijom za funkciju *mean()*. Ista stvar se može postići sa upisivanjem naredbe *?mean* (Braun & Murdoch, 2007).

Druga funkcija koja nam može pomoći je *example()* koja, također, za argument prima ime druge funkcije te izbacuje već ugrađeni primjer funkcije koja se traži (Braun & Murdoch, 2007). U dijelu koda ispod možemo vidjeti primjer *example()* funkcije za funkciju *mean()*.

```
> example(mean)


mean> x <- c(0:10, 50)


mean> xm <- mean(x)


mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

U primjeru za funkciju *mean()*, jezik R je unio neke vrijednosti u vektor te iz tog vektora izvukao srednju vrijednost (što zapravo radi *mean()* funkcija).

U slučaju da ne znamo ime funkcije za koju želimo pomoć onda unutar RStudia možemo pregledavati pomoć koju možemo pokrenuti sa naredbom *help.start()* (Braun & Murdoch, 2007).

4.9. Dodjeljivanje vrijednosti varijablama

Intuitivno za znak dodjeljivanja vrijednosti varijablama koristimo znak jednakosti, tj. „=“, međutim, iako s tim znakom možemo postići što želimo, u jeziku R najčešće se koristi znak „<-“ (Matloff, 2009). Istu funkciju ima i znak „->“ samo sa obrnutim redoslijedom (prvo ide vrijednost pa tek onda naziv varijable). Primjere dodjeljivanja vrijednosti možemo vidjeti u dijelu koda ispod.

```
> a=5
> a
[1] 5
> b<-10
> b
[1] 10
> 15->c
> c
[1] 15
```

Možemo vidjeti da smo sa sva 3 znaka postigli isto.

4.10. Tipovi podataka

Jezik R sadrži slične tipove podataka na koje smo navikli i u ostalim jezicima. Neki od tipova podataka su:

- *Numeric* (*Decimal*) – decimalni brojevi, zadano, svaki broj koji se unosi je ovaj tip podataka osim ako se ne označi drugačije,
- *Integer* – cijeli brojevi, da bi neki broj bio ovog tipa podataka, to se mora specificirati,
- *Complex* – kompleksni brojevi,
- *Logical* – logički tip podataka, u drugim jezicima se naziva boolean,
- *Character* – niz karaktera.

Ključnom riječi *mode* možemo dobiti koji je tip podatka neka varijabla te sa istom riječi možemo mijenjati tip podataka (R Core Team, 2019). Posebnost je samo što se i *numeric* i *integer* tipovi podataka prikazuju kao *numeric* te da bismo ih razlikovali, koristimo ključnu riječ *storage.mode*. Upravljanje tipovima podataka možemo vidjeti na primjeru dole.

```
> a<-5
> mode(a)
[1] "numeric"
> storage.mode(a)
[1] "double"
> storage.mode(a)<-"integer"     #mijenjamo tip podataka a u „integer“
> storage.mode(a)
[1] "integer"
> mode(a)
[1] "numeric"
> mode(a)<-"character"           #mijenjamo tip podataka a u „character“
> mode(a)
[1] "character"
```

Vidimo kako jednostavno možemo dodijeliti tip podataka nekoj varijabli. Na primjeru možemo, također, vidjeti kako *mode* ne vidi razliku između *integera* i *doubla*, ali zato možemo koristiti *storage.mode*.

4.11. Vektori

Vektori su osnovna struktura podataka u jeziku R, teško je zamisliti ikakav dio koda bez vektora (Matloff, 2009).

Ako bi uspoređivali sa vektore u jeziku R sa drugim jezicima, može se reći da vektori odgovaraju listama u drugim jezicima kao što su C ili C++. Čak ni skalari kao takvi ne postoje već su to vektori sa jednim elementom (Matloff, 2009).

Kao argumente, vektor ima duljinu i tip podataka (R Core Team, 2019). Jedan vektor može imati samo jedan tip podataka što znači da se u isti vektor ne mogu miješati npr. tipovi *numeric* i *character* (Matloff, 2009).

U odnosu na druge jezike, vektori u R-u su indeksirani tako da se počinje od indeksa 1 te se odatle inkrementira.

Postoji više načina da se vektorima dodjeljuju vrijednosti. Najjednostavnije je koristiti funkciju *c()* koja kombinira više vrijednosti u vektor. Druga opcija je da se vrijednost unosi tako što odredimo vektor u koji unosimo i indeks na koju poziciju unosimo. Tako, osim što možemo dodavati vrijednosti na kraj vektora, možemo i mijenjati već unesene vrijednosti. Oba primjera se mogu vidjeti na dijelu koda ispod.

```
> a<-c(3,6,9)
> a
[1] 3 6 9
> b[1]=4
> b
[1] 4
> b[2]=8
> b[3]=12
> b
[1]  4  8 12
> b[1]=3                    #mijenjamo vrijednost indeksu broj 1
> b
[1]  3  8 12
> a<-c("prvi","drugi",3)   #vektor sa character vrijednostima
> a
[1] "prvi"  "drugi" "3"
```

Na prvoj liniji možemo vidjeti kako se dodaju vrijednosti vektoru pomoću funkcije *c()*. Na linijama 4, 7 i 8 možemo vidjeti kako se dodaju vrijednosti u vektor te na liniji 11 možemo vidjeti kako se mijenja vrijednost vektora. Na liniji 14 vidimo kako se dodaju *character* vrijednosti u vektor. Kako jedan vektor može imati samo jedan tip podataka, vrijednost „3“ se zapisuje kao *character*, a ne kao *numerical*.

Bitno je napomenuti da je R osjetljiv na velika i mala slova tako da ako vektor sa nazivom *a* onda kasnije ne možemo pozivati taj vektor sa *A*. Isto vrijedi i za funkcije.

Da bismo lakše mogli čitati podatke, možemo dodijeliti nazive indeksima unutar vektora. To se može učiniti sa funkcijom *names()*. Korištenje te funkcije možemo vidjeti na sljedećem primjeru:

```
> a<-c(1,2,3)
> a
[1] 1 2 3
> names(a)<-c("prvi","drugi","treci")       #dodajemo nazive
> a
 prvi drugi treci
    1     2     3
```

R podržava osnovne operacije nad vektorima poput zbrajanja, množenja, korjenovanja i mnogih drugih. Neke od primjera operacija možemo vidjeti na dijelu koda ispod.

```
> a<-c(4,5,6)
> b<-c(3,2,1)
> a<-a+b                                #zbrajanje 2 vektora iste duljine
> a
[1] 7 7 7
> a*2                                   #množenje vektora skalarom
[1] 14 14 14
> sqrt(a)                               #korjenovanje vektora
[1] 2.645751 2.645751 2.645751
> a*c(1,2)                              #recikliranje vektora
[1]  7 14  7
Warning message:
In a * c(1, 2) :
  longer object length is not a multiple of shorter object length
> a[4]=8
> a
[1] 7 7 7 8
> a+c(2,4)                              #recikliranje vektora
[1]  9 11  9 12
```

Operacije između dva vektora se vrše tako da se vrše operacije vrijednosti na istim indeksima. U slučaju da dva vektora nemaju istu duljinu onda se manji vektor reciklira. To znači

da se vrijednosti ponavljaju, počevši od početka (Braun & Murdoch, 2007). U primjeru koda iznad, recikliranje je se dogodilo tri puta. Prvi put je kod množenja vektora skalarom gdje je taj skalar zapravo vektor sa jednim elementom te se taj element ponavlja za svaki element vektora s kojim se množi. Drugi primjer je množenje jednog vektora duljine 3 i drugog vektora duljine 2. Tu R izbaci upozorenje jer 3 nije djeljivo sa 2 pa se neki elementi u manjem vektoru ponavljaju više puta od drugih (vrijednost na indeksu 1 se dva puta množi, a vrijednost na indeksu 2 se jedan put množi). Zadnji primjer je zbrajanje dva vektora od kojih je jedan duljine 4, a drugi duljine 2. Pošto je 4 djeljivo sa 2, R ne izbacuje nikakvo upozorenje jer se svi elementi manjeg vektora jednako koriste.

Postoje dvije važne ugrađene funkcije koje se često koriste kod dodjeljivanja vrijednosti vektorima. To su *seq()* i *rep()*. Funkcija *seq()* generira aritmetičku sekvencu, dok funkcija *rep()* postavlja iste vrijednosti u vektor *n* puta (Matloff, 2009). Argumenti funkcije *seq()* su početak niza, završetak niza i vrijednost inkrementacije, a argumenti za funkciju *rep()* su vrijednosti i broj ponavljanja. Primjer ispod pokazuje korištenje obadvije funkcije.

```
> seq(10)                  #sekvenca bez argumenta početka niza
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(1,10)          #sekvenca bez argumenta vrijednosti inkrementacije
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(10,2)                #sekvenca sa negativnim inkrementiranjem
[1] 10  9  8  7  6  5  4  3  2
> seq(1,10,2)
[1] 1 3 5 7 9
> seq(1,2,0.1)
 [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
> rep(10,5)
[1] 10 10 10 10 10
> rep(5,10)
 [1] 5 5 5 5 5 5 5 5 5 5
> rep(seq(1,4),3)                      #kombiniranje seq() i rep()
 [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

Na primjeru vidimo razne mogućnosti korištenja ove dvije funkcije. Ako kod pozivanja funkcije *seq()* ne navedemo argument za početak niza ili vrijednost inkrementacije, onda su te vrijednosti jednake 1. Jezik R daje programeru toliku slobodu da je kao argument jedne funkcije moguće staviti drugu funkciju i to možemo vidjeti na liniji 15.

Jedna od najčešće korištenih operacija u jeziku R je filtriranje (Matloff, 2009). Filtriranje je izdvajanje elemenata nekog vektora ovisno o operaciji. Primjer filtriranja vektora možemo vidjeti ispod:

```
> a<-c(1,8,4,9)
> a>7
[1] FALSE  TRUE FALSE  TRUE
> b<-a[a>7]
> b
[1] 8 9
```

Na primjeru su iz vektora *a* prebačene sve vrijednosti veće od 7 u vektor *b*.

4.12. Višedimenzionalni vektori

Do sada je objašnjen koncept vektora i zaključeno je da su vektori sličan koncept kao i liste u drugim jezicima poput C++-a. Kao i u tim drugim jezicima, vektori mogu biti višedimenzionalni. U literaturi na engleskom jeziku, za višedimenzionalne vektori se pojam *array* koji smo, kod drugih jezika, navikli prevoditi sa pojmom lista, međutim, kako će se kasnije obrađivati drugi (dosta važniji) koncept koji prevodimo liste, za ovaj će se koristiti pojam višedimenzionalni vektori.

Funkcija koja se koristi za kreiranje višedimenzionalnih vektora je *array()* koja za argumente prima dva vektora, od kojih u jedan su smještene vrijednosti koje želimo da višedimenzionalni vektor primi, a u drugom su smještene vrijednosti dimenzija.

Najvažniji višedimenzionalni vektori su, naravno, oni sa dvije dimenzije i oni se nazivaju matrice te će se one obraditi malo kasnije (Matloff, 2009). Zasad možemo vidjeti vektor sa tri dimenzije prikazan na primjeru dole.

```
> a<-array(1:18, c(3,3,2))
> a
, , 1

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

, , 2
```

```
     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18

> a<-array(1:25, c(3,3,2))
> a
, , 1

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
> a[1,1,1]
[1] 1
> a[2,1,1]
[1] 2
> a[1,1,2]
[1] 10
> a[3,2,1]
[1] 6
> a[6]
[1] 6
```

U primjeru su kreirana dva vektora istih dimenzija. Razlika je u tome što su u prvi vektor smještene vrijednosti od 1 do 18 što se poklapa sa dimenzijama (3*3*2=18), a u drugi vrijednosti od 1 do 25 što se ne poklapa sa dimenzijama. Možemo vidjeti da je R zanemario sve vrijednosti koje premašuju okvire dimenzija te su ova dva vektora, na kraju, identična. R bi reciklirao vrijednosti ako smo unijeli broj vrijednosti koji je manji od okvira dimenzija.

Na kraju primjera imamo nekoliko dohvaćanja vrijednosti nekih elemenata višedimenzionalne matrice. Indeksi elementa se unose unutar jedne uglate zagrade te su odvojeni zarezom (za razliku od C++-a i sličnih jezika gdje se indeks svake dimenzije stavlja u zasebnu uglatu zagradu. Na istim primjerima možemo primijetiti da se, prilikom unošenja, prvi indeks inkrementira najbrže i kad dosegne svoj opseg, onda se inkrementira drugi indeks itd. (Braun & Murdoch, 2007). R dopušta da se u višedimenzionalni vektor indeksira kao jednodimenzionalni vektor pri čemu koristi isti princip kao kod unošenja (Braun & Murdoch, 2007). Takvo indeksiranje možemo vidjeti na liniji 40.

4.12.1. Matrice

Kao što je već navedeno, vektori sa dvije dimenzije se nazivaju matrice (Matloff, 2009).

Kako je matrica isto višedimenzionalni vektor, može se kreirati pomoću *array()* funkcije, ali ta funkcija nije praktična za kreiranje matrica jer, kako su matrice jednostavnije i korištenije od vektora sa tri ili više dimenzija, postoji funkcija *matrix()* koja se koristi u tu svrhu.

Primjer korištenja funkcije *matrix()* i svih njenih argumenata možemo vidjeti dole.

```
> a<-matrix(1:12, nrow=3, ncol=4)           #matrica bez ikakvih dodatnih
argumenata
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> a<-matrix(1:12, nrow=3, ncol=4, byrow=T)  #matrica sa argumentom byrow
> a
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> a<-matrix(1:12, nrow=3, ncol=4, dimnames=(list(c("prvi","drugi",
      "treci"),c("prva","druga","treca","cetvrta"))))   #matrica      sa
      argumentom dimnames
> a
      prva druga treca cetvrta
prvi     1     4     7      10
drugi    2     5     8      11
treci    3     6     9      12
```

```
> is.matrix(a)                          #provjera je li a matrica
[1] TRUE
```

Na primjerima možemo primijetiti neke sličnosti i neke razlike između funkcija *array()* i *matrix()*. Na obadvije funkcije se isto unose vrijednosti. Također, sličnost je ta da se na funkciji *matrix()* mogu definirati dimenzije pomoću jednog vektora kao kod funkcije *array()* ali radi lakšeg shvaćanja, bolje je koristiti argumente *nrow* i *ncol* pomoću kojih se definira broj redova, odnosno kolona.

Ono što je posebno za funkciju *matrix()* u odnosu na funkciju *array()* je argument *byrow* čije korištenje je prikazano na liniji 7. Prije je navedeno kako se višedimenzionalni vektori popunjavaju podatcima tako da se prvo popuni prvi indeks pa se tek onda inkrementiraju sljedeći indeksi. Argument *byrow* je specifičan za matrice koje su dvodimenzionalne te postavljanje njegove vrijednosti na *True*, prvo se popunjava drugi indeks koji predstavlja kolone pa se tek onda inkrementira prvi indeks koji predstavlja redove (R Core Team, 2019). Sa tim argumentom se dobiva transponirana matrica u odnosu na matricu koja bi se dobila bez tog argumenta.

Argumentom *dimnames* (linija 13) se mogu dodijeliti nazivi redovima i kolonama (kao i ostalim dimenzijama kod funkcije *array()*). Za dodjeljivanje naziva potrebno je koristiti koncept listi u koje se unose vektori sa željenim nazivima.

Na liniji 19 je korištena funkcija *is.matrix()* kojom se provjerava je li određeni (višedimenzionalni) vektor dvodimenzionalni, odnosno je li određeni vektor matrica (R Core Team, 2019).

4.13. Liste

Već je navedeno kako je R objektno orijentirani jezik. U tome važnu ulogu igra koncept liste (engl. Lists) koje možemo usporediti sa konceptima struktura u jeziku C++ i sličnima (Matloff, 2009).

Elementi liste mogu biti različitih tipova podataka te lista može spremati bilo koji drugi objekt (Braun & Murdoch, 2007).

Za kreiranje liste koristimo funkciju *list()* čiji su argumenti elementi koji se žele u listi. Ispod je prikazan dio koda gdje je kreirano nekoliko lista.

```
> a<-list(1,2,3)          #kreiranje liste bez naziva elemenata
> a
[[1]]
[1] 1
```

```
[[2]]
[1] 2

[[3]]
[1] 3

> a<-list(1,2,"3")         #kreiranje liste bez naziva elemenata
> a
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] "3"

> a<-list(prvi=1,drugi=2,treci="3")    #kreiranje   liste   sa   nazivima
      elemenata
> a
$prvi
[1] 1

$drugi
[1] 2

$treci
[1] "3"
> a[2]               #pozivanje jednog elementa liste
$drugi
[1] 2
> a$drugi                  #pozivanje jednog elementa liste pomoću naziva
[1] 2
```

Na dijelu koda je prikazano kreiranje tri liste. Prva lista (linija 1) se sastoji od 3 elementa od kojih je svaki *numerical* tip podatka, dok druga lista (linija 12) se sastoji od, također, 3 elementa, ali jedan element je *character* tip podatka. Tu možemo vidjeti kako se u istu listu

može staviti više elemenata sa različitim tipom podataka. Treća lista (linija 23) se sastoji od istih elemenata kao i druga ali su tu, na jako jednostavan način, dodijeljeni nazivi elementima.

Na linijama 33 i 36 možemo vidjeti dva primjera pozivanja jednog elementa liste. Prvi primjer je pomoću unesenog indeksa kao što je se radilo i u vektorima, a drugi primjer je pomoću naziva elementa pri čemu se koristi znak „$“.

Zbog svoje kompleksnosti, nećemo često kreirati liste, ali su mnoge funkcije vraćaju kompliciranе rezultate kao liste (Braun & Murdoch, 2007).

R dopušta da se, na već kreiranoj listi, dinamički dodavaju i brišu elementi. Primjer dodavanja i brisanja elemenata je prikazano na dijelu koda ispod.

```
> a<-list(prvi=1,drugi=2,treci="3")
> a
$prvi
[1] 1

$drugi
[1] 2

$treci
[1] "3"

> a$drugi<-NULL                 #brisanje elementa
> a
$prvi
[1] 1

$treci
[1] "3"

> a$cetvrti<-4                  #dodavanje elementa
> a
$prvi
[1] 1

$treci
[1] "3"
```

```
$cetvrti
[1] 4
```

Dodavanje elementa na već kreiranu listu (linija 20) funkcionira slično kao i sa vektorima, samo dodamo novi indeks te mu dodijelimo vrijednost. Brisanje elemenata (linija 12) funkcionira tako da nekom elementu dodijelimo vrijednost *NULL*.

4.13.1. Okviri podataka

Okviri podataka su liste čiji su elementi vektori jednake duljine. Može se reći da su kao matrice, međutim, razlika je u tome što kod okvira podataka, elementi mogu biti različitog tipa podataka (Matloff, 2009).

Velik dio operacija koje se mogu koristiti nad matricama, mogu se koristiti i nad okvirima podataka zbog slične strukture (Matloff, 2009).

Gotovo svi podatci koje se žele analizirati dolaze u više kolona pa su okviri podataka idealni za organizaciju tih podataka unutar jezika R (Braun & Murdoch, 2007) pa je to koncept koji se dosta koristi prilikom programiranja u R-u i koji će se dosta koristiti u ovome radu.

4.14. Programske strukture

R, kao i svi ostali programski jezici, sadrži programske strukture za upravljanjem slijeda izvršavanja.

4.14.1. Konstrukt *if()*

Kao i u većini drugih jezika, jedna od najvažnijih programskih struktura je *if()* konstrukt. Koncept je sličan kao i na tim drugim jezicima, tj. ako se zadovolji uvjet onda se izvršava tijelo ili ako se ne zadovolji uvjet, a dodan je *else* u konstrukt, onda se izvršava *else-a*.

Sintaksa *if()* konstrukta:

```
if (uvjet) {tijelo kad je uvjet zadovoljen}              #bez else
if (uvjet) {tijelo kad je uvjet zadovoljen} else {tijelo kad uvjet nije
      zadovoljen}          #sa else
```

Sintaksa je jednostavna i funkcionira kao u ostalim programskim jezicima. Ako je uvjet zadovoljen onda se izvršava tijelo, a ako nije, onda se ne izvršava ili se izvršava dio koda koji se nalazi u *else* dijelu.

4.14.2. Petlja *for()*

U jeziku R, *for()* petlja funkcionira slično kao i u većini drugih jezika (jednaka je jeziku Python).

Sintaksa za *for()* petlju je sljedeća:

```
for (nazivVarijable in vektor) {
   tijelo petlje
}
```

U svakoj iteraciji ove petlje, vrijednost varijable koja je nazvana *nazivVarijable*, se izjednači sa određenim elementom vektora koji je nazvan *vektor* te za svaku vrijednost vektora, izvršava se tijelo petlje (Braun & Murdoch, 2007).

Primjer korištenja for petlje možemo vidjeti u dijelu koda ispod.

```
a<-c(1:20)
b<-0
for(c in a){
  b=b+c
}
```

U ovom primjeru, *for()* petlja zbraja vrijednosti svih elemenata vektora sa nazivom *a*. Kako vektor *a* ima 20 elemenata, petlja ima 20 iteracija te se u svakoj iteraciji varijabli *c* dodijeli vrijednost trenutnog elementa vektora pa se s tom vrijednosti izvrši tijelo petlje.

Za sumiranje vrijednosti svih elemenata vektora, u R-u, postoji ugrađena funkcija *sum()* tako daje ova *for()* petlja nepotrebna pošto čini upravo to, ali poslužila je za primjer.

4.15. Funkcije

Funkcije u jeziku R rade na sličan princip kao u jezicima C++, Python i ostali. Kako R nema pokazivače, neke stvari su malo drugačije izvedene (Matloff, 2009).

Primjer jedne jednostavne funkcije možemo vidjeti na dijelu koda ispod.

```
f<-function(a){
  b<-0
  for (c in a){
    b=b+c
  }
  return(b)
```

```
}
```

Funkcija u ovome primjeru ima jedan argument koji je vektor i vraća sumu svih elemenata toga vektora.

Kod pozivanja, funkcija kreira svoje lokalne varijable (zanemaruje se ako se dvije varijable isto nazivaju) tako da se globalne varijable neće promijeniti osim ako nije navedeno da se mijenjaju a to se radi sa znakom „<<-“.

```
a<-5
f<-function(){
  a<-a*2
}
f()                    #varijabla a se ne mijenja
> a
[1] 5
g<-function(){
  a<<-a*2
}
g()                    #varijabla a se  mijenja
> a
[1] 10
```

Funkcija *f* kreira svoju lokalnu varijablu koja nema utjecaja na globalnu varijablu iako se isto nazivaju te zato globalna varijabla se ne mijenja. Funkcija *g* ne kreira lokalnu varijablu nego koristi globalnu te se globalna varijabla promijeni.

4.16.Čitanje

Unutar R sesije, moguće je čitati i pisati podatke pomoću raznih funkcija (Braun & Murdoch, 2007). Neke od tih funkcija će se obraditi u ovoj cjelini.

4.16.1. Čitanje s tipkovnice

Čitanje unosa korisnika sa tipkovnice se može ostvariti pomoću funkcija scan() i readline() (Matloff, 2009). Primjere korištenja te dvije funkcije vidimo na primjeru ispod.

```
> a<-scan()
1: 5                                  #korisnik unosi unutar konzole
2: 4 3                              #korisnik unosi unutar konzole
4: 2                                  #korisnik unosi unutar konzole
```

```
5: 1                                    #korisnik unosi unutar konzole
6:                                      #korisnik unosi unutar konzole
Read 5 items
> a
[1] 5 4 3 2 1
> a<-readline()
koristenje readline() funkcije          #korisnik unosi unutar konzole
> a
[1] "koristenje readline() funkcije"
```

Funkcija *scan()* služi za čitanje *numerical* vrijednosti. Funkcionira tako da korisnik unutar konzole upisuje vrijednosti sve dok ne ostavi unos prazan. Vrijednosti koje je korisnik unio se spremaju kao vektor. Funkcija *readline()* služi za čitanje *character* vrijednosti. Unos se sprema kao vektor, ali samo sa jednim elementom u kojeg je spremljen cijeli red koji je korisnik unio.

4.16.2. Čitanje iz datoteke

Čitanje iz datoteke se ostvaruje pomoću funkcije *read.table()* (Matloff, 2009). Spomenuta funkcija ima velik broj argumenata od koji su neki *file* (ime datoteke), *header* (ima li datoteka zaglavlje), *sep* (znak koji služi za razdvajanje podataka) (R Core Team, 2019). Ima još nekoliko argumenata, ali će se najčešće koristiti navedena tri.

Osim *read.table()* postoji još nekoliko funkcija koje služe u tu svrhu (Matloff, 2009). Najvažnije su svakako *read.csv()* koja učitava *.csv* datoteku (podaci su razdvojeni zarezom) i *read.xls()* koja učitava *.xls* datoteku.

4.16.3. Čitanje seta podataka za šah

Prvi korak analiziranja seta podataka za šah je svakako učitavanje datoteke. Već je spomenuto da je datoteka u *.csv* obliku pa je za čitanje najlakše koristiti *read.csv()* funkciju. Učitavanje datoteke možemo vidjeti na sljedećem primjeru:

```
sah<-read.csv(file="C:\\sah\\games.csv", header=TRUE)
```

U varijablu koja je nazvana *sah* je učitana datoteka kao okvir podatka. Od argumenata, osim obavezne putanje datoteke, stavili smo da datoteka ima zaglavlje jer je u jeziku R zadano da datoteka nema zaglavlje.

Prijašnjom naredbom je učitan okvir podataka sa 20058 unosa i svaki unos ima 16 kolona. Kako izgleda taj okvir podataka možemo vidjeti na sljedećem primjeru:

```
> sah
```

```
          id rated turns victory_status winner increment_code           w
      hite_id white_rating
1  TZJHLljE FALSE    13      outoftime  white          15+2           b
      ourgris         1500
2  l1NXvwaE  TRUE    16         resign  black          5+10
         a-00         1322
3  mIICvQHh  TRUE    61           mate  white          5+10
       ischia         1496
4  kWKvrqYL  TRUE    61           mate  white          20+0      daniam
      urashov         1439
5  9tXo1AUZ  TRUE    95           mate  white          30+3          ni
      k221107         1523
6  MsoDV9wj FALSE     5           draw   draw          10+0          tr
      elynn17         1250
7  qwU9rasv  TRUE    33         resign  white          10+0
      capa_jr         1520
8  RVN0N3VK FALSE     9         resign  black         15+30 daniel_like
      s_chess         1413
9  dwF3DJHO  TRUE    66         resign  black          15+0          eh
      abfanri         1439
10 afoMwnLg  TRUE   119           mate  white          10+0 daniel_like
      s_chess         1381
```

Radi jednostavnosti, možemo vidjeti samo 10 unosa i 8 kolona.

4.16.4. Uklanjanje nepotrebnih kolona iz okvira podataka

Kao što je navedeno ranije u ovome radu, nisu nam sve kolone potrebne. Nepotrebne kolone su *last_move_at* koja pokazuje kad je partija završena i *moves* koji sadrži sve poteze u partiji. Kolone se mogu jednostavno ukloniti sa sljedeće dvije naredbe:

```
sah$last_move_at<-NULL
sah$moves<-NULL
sah$id<-NULL
sah$rated<-NULL
sah$increment_code<-NULL
sah$opening_name<-NULL
sah$opening_ply<-NULL
```

Nakon izvršenja naredbi, okvir podataka ima 14 kolona.

4.17.Nacrti u R

Da bi se napravio nacrt u jeziku R, korisnik poziva jednu od mnogih funkcija koje, ili kreiraju cijeli novi nacrt, ili dodaju neku stvar na postojeći nacrt (Murrel, 2006). Prvi tip funkcija

su funkcije visoke razine (engl. High level plots), a drugi funkcije niske razine (engl. Low level plots).

4.17.1. Funkcija *plot()*

Funkcija *plot()* je funkcija visoke razine kojom dobivamo jednostavni koordinatni sustav (engl. scatterplot). Najvažniji argumenti ove funkcije su naravno podaci od kojih će se napraviti grafikon. To su jedan ili dva vektora iste duljine (ako je jedan onda se za drugi vektor uzimaju indeksi prvog) (Braun & Murdoch, 2007). Drugi bitan argument je *type* kojim se definira tip grafa. Neke od vrijednosti toga argumenta su:

- „p“ – zadana vrijednost, podaci se na grafu označavaju točkama
- „l“ – podaci se označavaju linijom
- „b“ – podaci se označavaju i linijom i točkama
- „o“ – slično kao „b“, samo se linija ne prekida u blizini točke (prolazi kroz nju)
- „h“ – podaci se označavaju u obliku histograma (4.17.9. Primjer: postotak pobjeda bijelog igrača ovisno o načinu otvaranja) (R Core Team, 2019)

Ostali bitni argumenti ove funkcije su *main* (naslov grafikona), *sub* (podnaslov grafikona), *xlab* (naziv *x* osi), *ylab (*naziv *y* osi) (R Core Team, 2019).

Primjer korištenja ove funkcije može se vidjeti u cjelini 3.17.6. (*Primjer: odnos broja poteza i pobjednika*), 3.17.7. (Primjer: odnos razlike u ocjenama igrača i broja poteza), 3.17.9. (Primjer: povijest igrača)

4.17.2. Funkcija *hist()*

Funkcija *hist()* je funkcija visoke razine kojom dobivamo histogram. Histogram je grafikon koji prikazuje distribuciju podataka unutar određenog intervala (Braun & Murdoch, 2007).

Kao i kod funkcije *plot()*, najvažniji argumenti su podaci. Karakteristično za histograme, važan argument je *breaks* pomoću kojega se definira raspon ćelija unutar grafa. U R-u je zadano da sve ćelije imaju jednak raspon (R Core Team, 2019).

Sa funkcijom *plot()*, funkcija *hist()* dijeli neke argumente kao što su *main*, *sub*, *xlab*, *ylab* (R Core Team, 2019).

Primjer korištenja funkcije *hist()* se može vidjeti u cjelini 3.17.5. (Primjer: raspored ocijene bijelog igrača)

4.17.3. Funkcija *pie()*

Funkcija *pie()* je funkcija visoke razine kojom dobivamo kružni grafikon (engl. pie chart). Kružni grafikon prikazuje podijeljenost neke cjeline na kružne isječke.

Kružni grafikon je malo drugačiji od histograma i običnog grafikona, ali i dalje glavni argument je vektor koji predstavlja podatke. Drugi bitan argument je *labels* pomoću kojega svakom kružnom isječku možemo dodijeliti naziv.

Postoje još mnogi argumenti koji služe za uređivanje izgleda kružnog grafikona kao što je *radius*, *clockwise* (orijentacija prikazivanja podataka), *col* (boje isječaka grafikona) i mnogi drugi.

Primjer korištenja ove funkcije može se vidjeti u cjelini 3.17.4 (*Primjer: odnos neriješenih partija i pobjeda bijelog i crnog igrača*)

4.17.4. Primjer: odnos neriješenih partija i pobjeda bijelog i crnog igrača

Može se reći da je, u setu podataka sa šahovskim partijama, najvažniji ishod partije. Samim time, prvi primjer analize seta podataka je jednostavni kružni grafikon koji pokazuje odnos neriješenih partija te pobjeda crnog ili bijelog igrača.

Kolona *winner* je *character* tipa te, u svim zapisima, može imati vrijednost „white“ (bijeli igrač je pobjednik), „black“ (crni igrač je pobjednik), „draw“ (neriješeno).

Najjednostavniji način za kreiranje jednostavnog kružnog grafikona je funkcija *pie()*. U ovome primjeru, argumenti funkcije koji se koriste su vektori sa brojem svakog ishoda i nazivi svakog ishoda, boja isječaka te naslov grafikona.

Kod za dobivanje ovog kružnog grafikona:

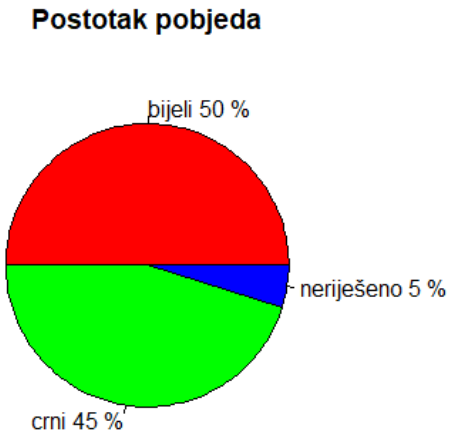
```
pobjede<-c(0,0,0)       #vektor u koji će biti spremljen broj pobjeda
lblPobjede<-
c("bijeli", "crni", "izjednačeno")  #vektor u koji su spremljeni nazivi

#analiziranje svake partije:
for(i in 1:20058){
  if(sah$winner[i]=="white"){             #ako je bijeli dobio
    pobjede[1]<-pobjede[1]+1
  }else if(sah$winner[i]=="black"){             #ako je crni dobio
    pobjede[2]<-pobjede[2]+1
```

```
  }else if(sah$winner[i]=="draw"){              #ako je izjednačeno
    pobjede[3]<-pobjede[3]+1
  }
}
#dodavanje postotka na naziv:
lblPobjede[1]<-
paste(lblPobjede[1],round(100*pobjede[1]/sum(pobjede)),"%")
lblPobjede[2]<-
paste(lblPobjede[2],round(100*pobjede[2]/sum(pobjede)),"%")
lblPobjede[3]<-
paste(lblPobjede[3],round(100*pobjede[3]/sum(pobjede)),"%")
pie(pobjede, labels=lblPobjede, col=rainbow(length(lblPobjede)),main="Po
      stotak pobjeda")     #pravljenje kružnog grafikona
```

Podaci koji se trebaju nalaziti na grafikonu se smještaju u vektor *pobjede* kojem je duljina 3. U *for()* petlji se pregledava vrijednost svakog zapisa te ovisno o vrijednosti, povećava se određeni element vektora *pobjede*.

Funkcijom *paste()*, na vektor *lblPobjede*, u kojem se smještaju nazivi isječaka grafikona, se doda postotak koji će kasnije bit prikazan na grafikonu.

Grafikon se može vidjeti na slici 3.



Slika 3. odnos neriješenih partija i pobjeda bijelog i crnog igrača

Obzirom na prednost početnog poteza, za očekivati je da bijeli igrač ima veći postotak pobjeda od crnog. Bijeli igrač je pobjeđivao u 50 posto partija, dok je crni u 45 posto.

Na stranici *lichess.org*, jedini način da partija završi neriješeno je da jedan od igrača ponudi opciju *draw* te da je drugi igrač prihvati što znači da se neriješene partije događaju relativno rijetko i to možemo vidjeti na grafikonu.

4.17.5. Primjer: raspored ocijene bijelog igrača

U setu podataka, jedna kolona predstavlja ocjenu bijelog igrača. Ta kolona je *numerical* tipa.

Raspored ocijene bijelog igrača se najjasnije vidi na histogramu koji se najlakše dobiva funkcijom *hist()*.

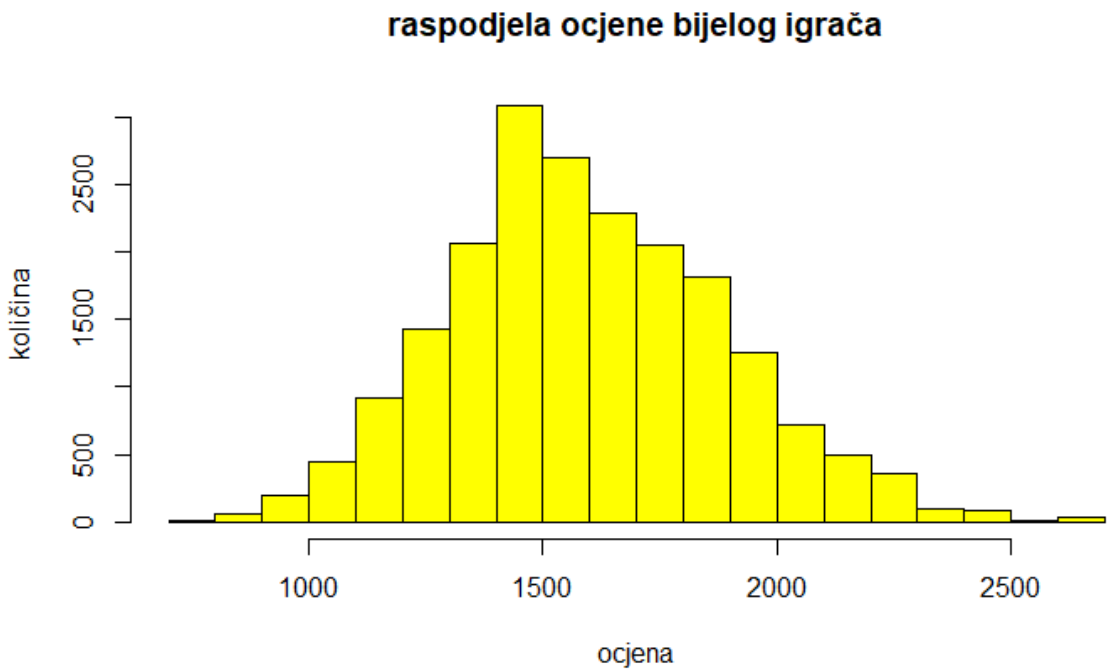
Ovaj histogram je jako jednostavan te se može dobiti sljedećom linijom koda:

```
hist(sah$white_rating, col="yellow", xlab="ocjena", ylab="količina", mai
      n="raspodjela ocjene bijelog igrača")
```

Grafikon koji se dobije navedenom linijom koda se može vidjeti na slici 4.

Na *x* osi su prikazani intervali ocjene bijelog igrača, a na *y* osi je prikazan broj bijelih igrača kojima je ocjena unutar nekog intervala.

Na grafikonu se vidi da je najčešća ocjena u intervalu od 1400 do 1500 te se ta ocjena ponavlja oko 2750 puta, sljedeći najčešći interval je od 1500 do 1600 koji se ponavlja oko 2500 puta. Također, može se vidjeti da, što je dalje ocjena od 1500, to se manje puta pojavljuje.



Slika 4. raspodjela ocjene bijelog igrača

4.17.6. Primjer: odnos broja poteza i pobjednika

Jedna kolona u setu podataka je *turns* te označava koliko je poteza bilo potrebno da se partija završi. Navedena kolona je *numerical* tipa te uz postotak pobjeda bijelog igrača tvori osi koordinatnog sustava.

Koordinatni sustav je najlakše dobiti funkcijom *plot()*. Osim podataka, u argumentima se nalazi i naslov grafikona te nazivi svake osi.

Obzirom da su partije sa preko 200 poteza rijetke, one se isključuju iz analize jer ne pokazuju realno stanje. Također, partije bez ijednog poteza, koje se mogu dobiti samo predajom, se isključuju jer ne ovise o boji igrača.

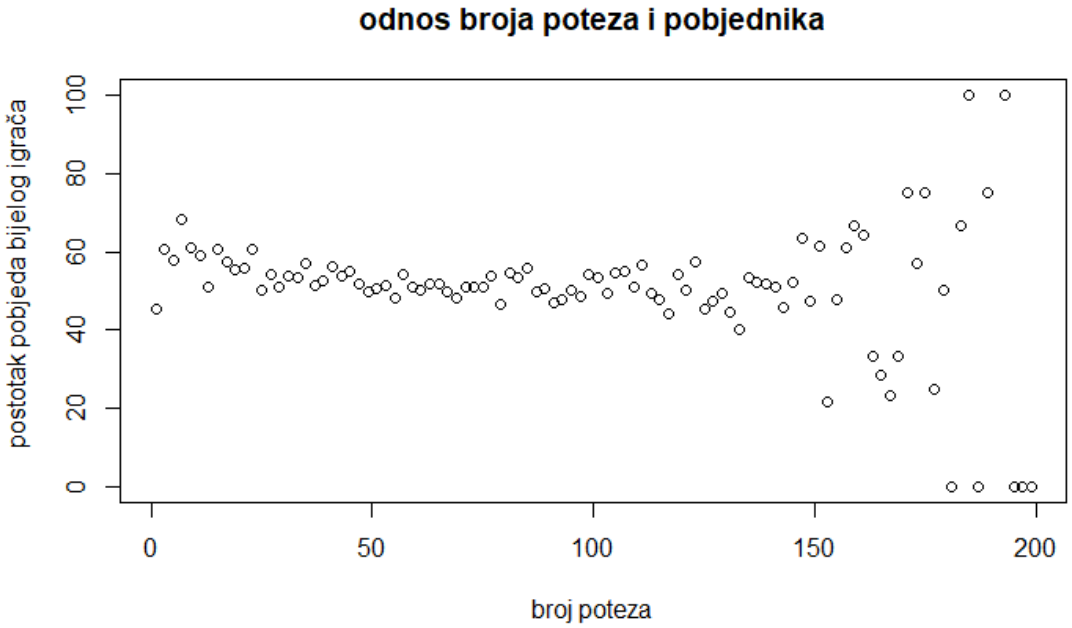
Kod za dobivanje grafikona:

```
bijeli<-rep(0,200)                    #broj pobjeda bijelog igrača

crni<-rep(0,200)                    #broj pobjeda bijelog igrača

#analiziranje svake partije:

for(i in 1:20058){

  if(sah$turns[i]<=200 &sah$turns[i]>=1){           #ne računaju se part
      ije bez ijednog poteza i sa više od 200 poteza

    if(sah$winner[i]=="black"){

      crni[sah$turns[i]]=crni[sah$turns[i]]+1     #ako je crni pobijedio


    }else if(sah$winner[i]=="white"){

      bijeli[sah$turns[i]]=bijeli[sah$turns[i]]+1
      #ako je bijeli pobijedio

    }

  }

}

#računanje postotka pobjeda bijelog igrača u vektor odnos

odnos<-NULL

for(i in seq(1,200,2)){

  odnos=c(odnos,100*(bijeli[i]+bijeli[i+1])/((bijeli[i]+bijeli[i+1])+(cr
      ni[i]+crni[i+1])))

}

plot(seq(1,200,2),odnos, main="odnos broja poteza i pobjednika", xlab="b
      roj poteza", ylab="postotak pobjeda bijelog igrača")
         #pravljenje grafikona
```

Vektori *bijeli* i *crni* imaju duljinu 200 te se je na početku svaki element jednak 0 (ostvareno pomoću *rep()* funkcije). Za svaku partiju (koja ima više od 0, manje od 201 poteza i nije neriješena), ovisno o pobjedniku, povećava se element jednog od navedena dva vektora sa indeksom koji je jednak broju poteza partije.

Nakon pregledanih svih partija, u vektor *odnos* se upisuje postotak pobjeda bijelog igrača. Kako je uvijek bijeli igrač prvi na redu, onda će uvijek bijeli igrač igrati na neparnom rednom broju poteza, samim time, nakon neparnog broja poteza, pobjednik mora biti bijeli igrač (osim ako bijeli igrač preda). Iz tog razloga, vektor *odnos* nije duljine 200, nego duljine 100 te se u jedan element zapisuje prosjek dva elementa vektora *bijeli* i *crni* (npr. broj poteza koji iznosi 11 se sprema sa brojem poteza koji iznosi 12).

Dobiveni grafikon se može vidjeti na slici 5.



Slika 5. odnos broja poteza i pobjednika

Zbog malog broja partija, podaci sa preko 150 poteza ne daju realan rezultat pa su zato podaci ovako oscilirajući. Analizirati će se samo podaci sa ispod 150 poteza.

Kako bijeli igrač ima prednost početnog poteza, ta prednost je izraženija sa manjim brojem poteza. To se vidi na grafu pa je sa manjim brojem poteza, postotak pobjeda bijelog igrača čak 60 posto, dok sa većim brojem poteza (oko 100) se prednost smanjuje i približava se 50 posto.

4.17.7. Primjer: odnos razlike u ocjenama igrača i broja poteza

U setu podataka, postoje dvije kolone koje označavaju ocjenu bijelog, odnosno crnog igrača. To su kolone *white_rating* i *black_rating*. Te dvije kolone su *numerical* tipa.

Ocjena igrača se računa prema rezultatima igrača protiv drugih igrača te, intuitivno, što je veća ocjena, to bi igrač trebao biti bolji.

Za kreiranje koordinatnog sustava se koristi funkcija *plot()* kojem je, u ovom primjeru, novost argument *cex* koji služi za mijenjanje veličine točke na grafikonu. Kako ovaj grafikon sadrži velik broj točaka, veličina istih se treba smanjiti.

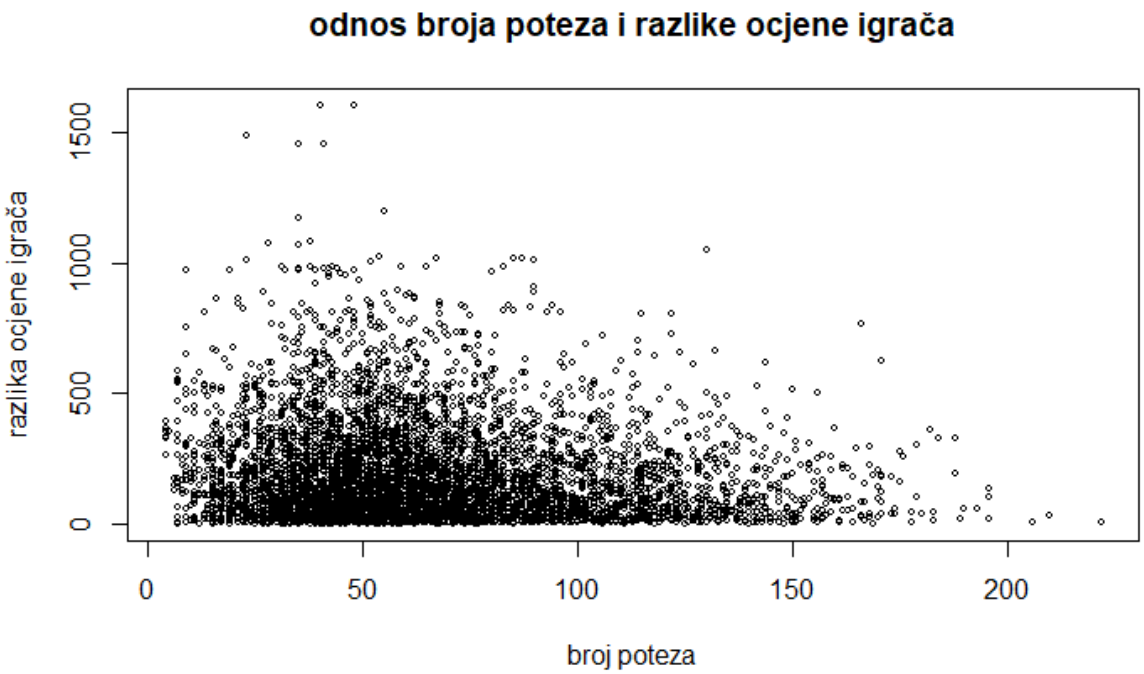
Kod za dobivanje grafikona:

```
potezi<-NULL                          #broj poteza
razlika<-NULL                         #razlika ocjene igrača
#analiziranje svake partije:
for(i in 1:20058){
  if(sah$victory_status[i]=="mate"){  #partija se uzima u obzir samo ako
      je završena matom
    potezi<-c(potezi,sah$turns[i])          #dodavanje broja poteza u
      vektor
    if(sah$white_rating[i]>sah$black_rating[i]){
      razlika<-c(razlika,(sah$white_rating[i]-sah$black_rating[i]))
         #dodavanje razlike u vektor
    }else{
      razlika<-c(razlika,(sah$black_rating[i]-sah$white_rating[i]))
         #dodavanje razlike u vektor
    }
  }
}
plot(potezi,razlika,cex=0.5,main="odnos broja poteza i razlike ocjene ig
      rača",xlab="broj poteza",ylab="razlika ocjene igrača")
         #pravljenje grafikona
```

Podaci koji će se skicirati na grafikonu su smješteni u vektore *potezi* i *razlika.* Kako se ne zna kolika će biti duljina tih vektora, oni su na početku postavljeni na *NULL* vrijednost te se svakim ponavljanjem *for()* petlje popunjavaju podacima.

Unutar funkcije *plot()*, koristi se argument *cex* kojim se mijenja veličina točke na grafikonu. Ako je vrijednost jednaka 1 onda su točke zadane veličine (može se vidjeti na slici 5.) te da bi smanjili tu veličinu, pošto ovaj grafikon ima velik broj točaka, vrijednost je postavljena na 0.5.

Grafikon se može vidjeti na slici 6.



Slika 6. odnos razlike u ocjenama igrača i broja poteza

Kako ocjena igrača reprezentira kvalitetu tog igrača, pretpostavka je da sa većom razlikom ocjene (veća razlika u kvaliteti) će biti manje poteza na partiji.

Na grafikonu se vidi da je većina partija imala oko 50 poteza te razlika u ocjeni igrača je manja od 500. Međutim, na grafikonu se vidi da partije sa najvećim brojem poteza (preko 200) su bile između igrača sa minimalnom razlikom u ocjeni. Isto tako, ako je razlika u ocjeni igrača bila velika (preko 1000) onda je ta partija završila sa relativno malim brojem poteza.

4.17.8. Primjer: postotak pobjeda bijelog igrača ovisno o načinu otvaranja

Među kolonama u setu podataka se nalazi kolona *opening_eco* koja je *character* tipa. U njoj je zapisan kod otvaranja. Kodovi su standardizirani te svako otvaranje ima svoj kod. Npr. jedan od poznatijih otvaranja je sicilijanska obrana i njezine varijacije imaju kodove u rasponu od „B20“ do „B99“ [3].

[3] https://www.365chess.com/eco.php

Grafikon koji prikazuje postotak pobjeda bijelog igrača ovisno o načinu otvaranja je najjednostavnije realizirati sa *plot()* funkcijom čiji je tip histogram („h“).

Za ovu analizu se koriste samo otvaranja koji se ponavljaju preko 100 puta te radi toga se ovaj grafikon može smatrati vjerodostojnim pokazateljem.

Kod za kreiranje grafikona:

```
otvaranje<-NULL                  #kod otvaranja
bijeli<-NULL                     #broj pobjeda bijelog igrača
ukupno<-NULL                     #broj partija
#analiziranje svakog zapisa:
for(i in 1:20058){
  if(as.character(sah$opening_eco[i]) %in% otvaranje){#ako  je  otvaranje
      već zapisano
    if(sah$winner[i]=="white"){        #ako je pobjednik bijeli igrač
      bijeli[match(sah2$opening_eco[i],otvaranje)]=bijeli[match(sah2$ope
      ning_eco[i],otvaranje)]+1
      ukupno[match(sah2$opening_eco[i],otvaranje)]=ukupno[match(sah2$ope
      ning_eco[i],otvaranje)]+1
    }else if(sah$winner[i]=="black"){       #ako je pobjednik crni igrač
      ukupno[match(sah2$opening_eco[i],otvaranje)]=ukupno[match(sah2$ope
      ning_eco[i],otvaranje)]+1
    }
  }else{                         #ako otvaranje nije zapisano
    otvaranje<-c(otvaranje,as.character(sah$opening_eco[i]))
    ukupno<-c(ukupno,1)
    if(sah$winner[i]=="white"){
      bijeli<-c(bijeli,1)
    }else{
      bijeli<-c(bijeli,0)
    }
  }
}
#eliminiranje otvaranja koji su bili manje od 101 put:
otvaranje2<-NULL
bijeli2<-NULL
ukupno2<-NULL
for(i in 1:length(otvaranje)){
  if(ukupno[i]>100){
```

```
    ukupno2<-c(ukupno2,ukupno[i])
    bijeli2<-c(bijeli2,bijeli[i])
    otvaranje2<-c(otvaranje2,otvaranje[i])
  }
}
#sortiranje
omjer=100*bijeli2/ukupno2
omjer=omjer[order(-omjer)]
otvaranje2=otvaranje2[order(-omjer)]
plot(omjer,   type="h",   lwd=21,   col=topo.colors(length(otvaranje2)),
     ylim=c(0,100), main="postotak pobjeda bijelog igrača ovisno o načinu
     otvaranja",       ylab="omjer pobjeda bijelog igrača", xlab="kod
     otvaranja")          #kreiranje grafikona
text(omjer, labels=otvaranje2, pos=3, cex=0.7)  #dodavanje naziva kodova
```

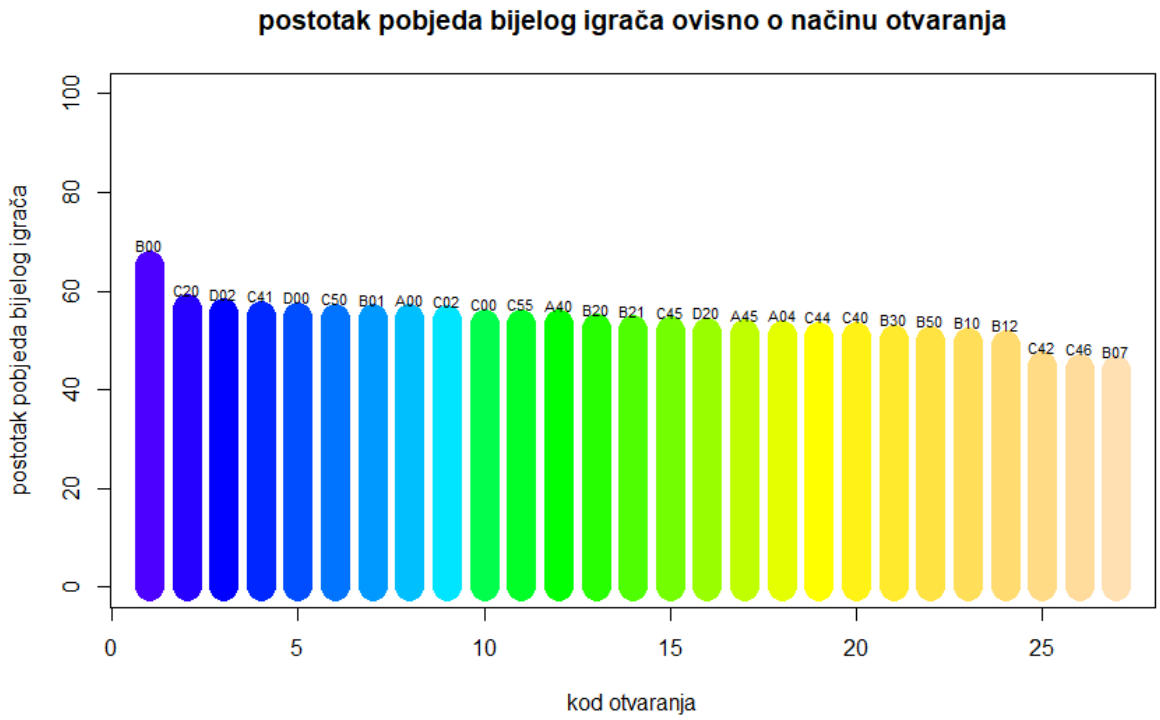
Na početku se definiraju tri vektora, to su *otvaranje* (kod otvaranja), *bijeli* (broj pobjeda bijelog igrača prilikom nekog otvaranja) i *ukupno* (broj partija prilikom nekog otvaranja).

Petlja prilikom koje se analiziraju svi zapisi funkcionira tako da se provjeri je li kod trenutne iteracije već unesen u vektor *otvaranje*. Ako jest onda se, ovisno o ishodu partije, inkrementira element vektora *bijeli* i *ukupno*, a ako nije ona se dodaje novi element na sva 3 vektora.

Idući korak je eliminiranje svih otvaranja koji su se ponovili manje od 101 put. To je realizirano tako da se kreiraju nova tri vektora u koji se zapisuju samo elementi početna tri vektora koji su se ponovili više od 100 puta. Nakon toga, vektori se sortiraju da na grafikonu budu prikazani padajućim redoslijedom.

Funkcija *plot()*, u ovom primjeru, je tipa „h“ što znači da kreira histogram. Funkcija *text()* služi za dodavanje teksta na grafikon i ovdje se koristi za imenovanje ćelija.

Grafikon se može vidjeti na slici 7.



Slika 7. postotak pobjeda bijelog igrača ovisno o načinu otvaranja

Na grafikonu se može vidjeti da je, bijelome igraču, najuspješnije otvaranje „B00“ te je postotak pobjeda bijelog igrača približno 70 posto kad je odigrano to otvaranje. S druge strane, po bijelog igrača, najmanje uspješno otvaranje je „B07“ kod kojega je postotak pobjeda oko 45 posto.

4.17.9. Primjer: povijest igrača

U setu podataka, postoje kolone *white_id* i *black_id* u kojima su spremljena korisnička imena igrača. Te dvije kolone su *character* tipa.

U ovome primjeru, koristi se i kolona *created_at*, koja je *numerical* tipa i u kojoj je zapisano vrijeme početka partije i služi za sortiranje partija obzirom da nisu sortirane u setu podataka.

Grafikon se realizira pomoću *plot()* i *segments()* funkcija. Funkcija *segments()* služi za crtanje linija na grafikonu.

Za analiziranje njegove povijesti, u obzir je uzet korisnik sa korisničkim imenom „totti“. Uzet je iz tog razloga što, u setu podataka, postoji velik broj njegovih partija (20 partija).

```
ocjena<-NULL                    #ocjena igrača
```

```
pobjeda<-NULL                    #je li igrač pobijedio partiju
pocetak<-NULL                    #vrijeme početka partije
#analiziranje svakog zapisa:
for(i in 1:20058){
  if(sah$white_id[i]=="totti"){
    ocjena<-c(ocjena, sah$white_rating[i])
    pocetak<-c(pocetak, sah3$created_at[i])
    if(sah$winner[i]=="white"){
      pobjeda<-c(pobjeda, T)
    }else{
      pobjeda<-c(pobjeda, F)
    }
  }else if(sah$black_id[i]=="totti"){
    ocjena<-c(ocjena, sah$black_rating[i])
    pocetak<-c(pocetak, sah3$created_at[i])
    if(sah$winner[i]=="black"){
      pobjeda<-c(pobjeda, T)
    }else{
      pobjeda<-c(pobjeda, F)
    }
  }
}
#sortiranje partija:
ocjena<-ocjena[order(pocetak)]
pobjeda<-pobjeda[order(pocetak)]
pocetak<-pocetak[order(pocetak)]
plot(ocjena, main="povijest igrača 'totti'", xlab="broj partije", ylab="
      ocjena igrača")            #kreiranje grafikona
#crtanje linija
for(i in 2:length(pobjeda)){
  if(pobjeda[i-1]==T){
    segments(i-1,ocjena[i-1],x1=i,y1=ocjena[i], col="green")
        #crtanje linije ako je igrač pobijedio partiju
  }else{
    segments(i-1,ocjena[i-1],x1=i,y1=ocjena[i], col="red")    #crtanje
      linije ako je igrač izgubio partiju
  }
```

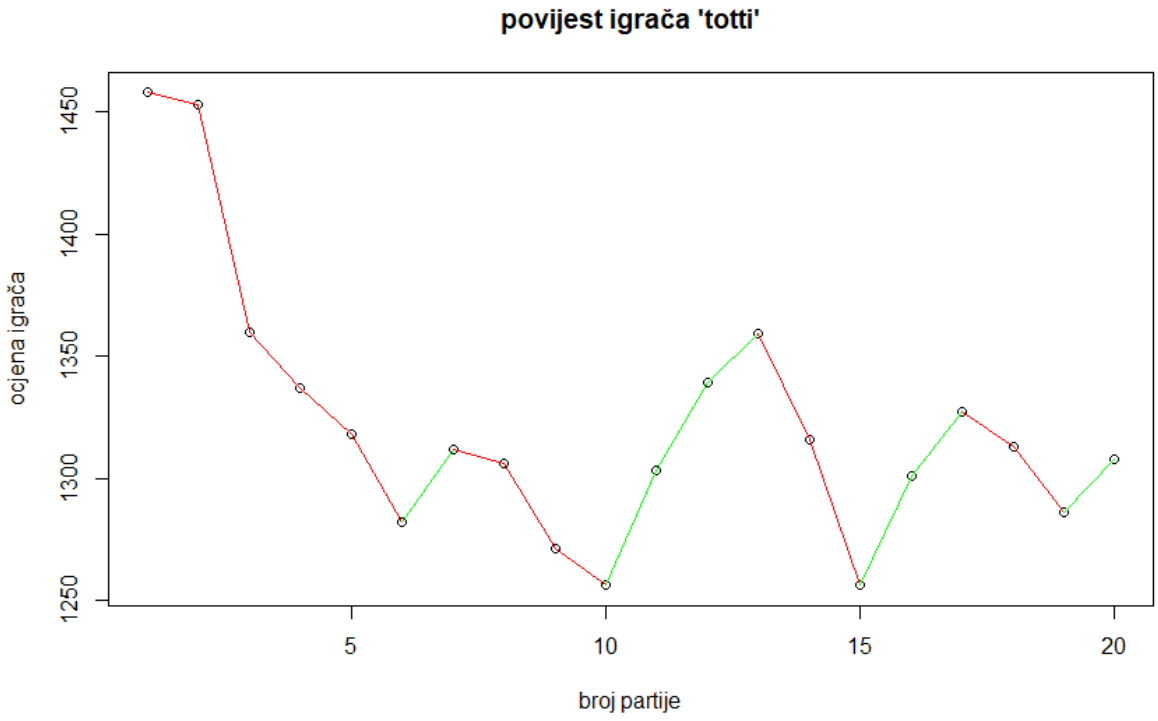
```
}
```

Za kreiranje grafikona, potrebna su tri vektora. To su *ocjena* (ocjena igrača uoči partije), *pobjeda* (je li igrač pobijedio partiju), *pocetak* (vijeme početka partije). Navedena tri vektora se popunjavaju podacima u slučaju da je vrijednost kolona *white_id* ili *black_id* jednaka „totti“.

Obzirom da zapisi u setu podataka nisu poredani kronološki, potrebno je sortirati partije pomoću vektora *početak*.

Nakon sortiranja, kreira se grafikon sa točkama u visini ocjene igrača pa se funkcijom *segments()* koja je funkcija niske razine crtaju linije kojima se spajaju točke te je linija, ovisno o ishodu partije, crvene ili zelene boje. Slična stvar je se mogla postići da je funkcija *plot()* bila tipa „o“, ali bi sve crte bile iste boje.

Grafikon se može vidjeti na slici 8.



Slika 8. povijest igrača

Na grafikonu se vidi kako je ocjena igrača mijenjala obzirom na ishod partije. Ovisno o ocjeni protivnika, ocjena se može više ili manje rasti ili padati što se može vidjeti na grafikonu.

5. Zaključak

U ovome radu je se objasnio pojam programskih jezika svojstvenih za problemsku domenu (DSL). Jezici koji se mogu smatrati DSL-ovima mogu značajno pojednostaviti posao programeru u vezi nekih stvari koje spadaju u domenu tog jezika, međutim, to ne znači da se ti jezici uvijek trebaju koristiti.

Kao primjer DSL-a, uzet je programski jezik R koji je namijenjen uglavnom za statističare. R, kao jedan od najkvalitetnijih alata namijenjenih za statističku analizu, je besplatan i lako dostupan. Isto važi i za njegovo najpopularnije razvojno okruženje, RStudio. Velika prednost jezika R je definitivno činjenica da redovno dobiva ažuriranja i postoje mnoge biblioteke koje čine ovaj jezik konkurentnim u usporedbi sa ostalim jezicima.

U ovom radu je objašnjena sintaksa jezika programskog R koja je slična ostalim poznatim jezicima kao što je C ili Python te način na koji R sprema podatke koji nije sličan navedenim jezicima. Tako, glavna struktura u većini ostalih jezika, obična varijabla sa jednim elementom kao takva ne postoji u R-u nego je glavna struktura vektor koji se može usporediti sa poljima u navedenim jezicima. U radu su objašnjene i ostale strukture podataka kao što su višedimenzionalni vektori (i matrice) i liste. Konstrukti i petlje funkcioniraju isto kao u ostalim jezicima.

Nacrti i grafikonu u jeziku R, kao jedna od njegovih najvećih snaga, također su obrađeni. Od široke palete ugrađenih funkcija koje se koriste u R-u, navedeno ih je nekoliko jednostavnih koje se najčešće koriste. Također, R ima velik broj biblioteka koje se mogu koristiti u svrhu pravljenja nacrta i grafikona. U ovome radu su se koristile samo one jednostavnije funkcije da se prikaže snaga ovog jezika, ali R ima još velik broj drugih funkcija.

R je pogodan za analiziranje raznih podataka što je u ovom radu prikazano kroz set podataka sa šahovskim partijama.

Analizom seta podataka sa šahovskim partijama u jeziku R, kroz nekoliko primjera, pokazano je nekoliko stvari. Tako, npr. znamo da bijeli ima veće šanse da pobjedi crnog igrača (4.17.4. Primjer: odnos neriješenih partija i pobjeda bijelog i crnog igrača), ali sa većim brojem poteza, šanse se smanjuju (4.17.6. Primjer: odnos broja poteza i pobjednika). Također, pokazano je da razlika u ocjeni igrača utječe na broj poteza u partiji (4.17.7. Primjer: odnos razlike u ocjenama igrača i broja poteza), pokazano je uz koja otvaranja bijeli igrač ima najveće šanse za pobjedu (4.17.8. Primjer: postotak pobjeda bijelog igrača ovisno o načinu otvaranja) te je grafički prikazana povijest jednog igrača gdje možemo vidjeti kako pobjeda ili poraz utječu na ocjenu igrača (4.17.9. Primjer: povijest igrača).

Literatura

- Braun, W. J., & Murdoch, D. J. (2007). *A First Course in Statistical Programming with R*. Retrieved from http://einspem.upm.edu.my/wopr2017/2016.pdf
- IBM. (2012). *Deep Blue*. Retrieved from https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/
- Martin Fowler. (2010). Domain-Specific Languages. In *Addison-Wesley Professional* (Vol. 13). Retrieved from http://sagierp.com.br/devel/cursos/Agile/Domain Specific Languages.pdf
- Matloff, N. (2009). *The Art of R Programming*. Retrieved from http://heather.cs.ucdavis.edu/~matloff/132/NSPpart.pdf
- Mernik, M., Heering, J., & Sloane, A. M. (2011). *When and How to Develop Domain-Specific Languages*. *37*(4), 699–709. https://doi.org/10.1115/ipc2010-31470
- Murrel, P. (2006). *R Graphics*. Retrieved from http://lux.e-reading.bz/bookreader.php/137370/C486x_C04.pdf
- r-project.org. (2019). A language and environment for statistical computing. Retrieved from https://www.r-project.org/
- R Core Team. (2019). *An introduction to R*. Retrieved from https://cran.r-project.org/doc/manuals/r-release/R-intro.html
- RStudio, I. (2019). *RStudio*. Retrieved from https://www.rstudio.com/products/rstudio/
- van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, *35*(6), 26–36. https://doi.org/10.1145/352029.352035

Popis slika