UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Jovan Dmitrović

MASTER IZ MATEMATIKE ILI RAČUNARSTVA ČIJI JE NASLOV JAKO DUGAČAK

master rad

Beograd, 2021.

Mentor:

dr Mika Mikić, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Ana Anić, vanredni profesor
University of Disneyland, Nedođija

dr Laza Lazić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami, tati i dedi

Naslov master rada: Master iz matematike ili računarstva čiji je naslov jako dugačak

Rezime: Apstrakt ide ovde

Ključne reči: programiranje, programski jezici

Sadržaj

1 Uvod **1**

2 Programski jezik *Rust* **2**

2.1 Razvoj jezika *Rust* 2

2.2 Instalacija i korišćenje sistema *Cargo* 3

2.3 Osnovne karakteristike jezika 5

3 Razrada **15**

4 Zaključak **16**

Literatura **17**

Glava 1

Uvod

Glava 2

Programski jezik *Rust*

Rust je statički tipiziran jezik fokusiran na bezbednost i performanse. Od svog nastanka, ovaj jezik je dobio veliku pažnju u svetu programiranja, čemu svedoči i činjenica da je *Rust* bio proglašen za „omiljeni programski jezik" već petu godinu za redom u anketi koju je sprovela popularna veb-stranica *Stack Overflow* [11].

Danas se *Rust* koristi na velikom broju ozbiljnih projekata. Na primer:

- AWS servisima firme Amazon, poput *Lambda*, *EC2* i *Cloudfront* [2],
- U okviru operativnog sistema kompanije Gugl (engl. *Google*) *ChromeOS* [3],
- Određenim komponentama Majkrosoftove platforme *Azure*, uključujući i IoT sigurnosni servis *edgelet* [5],
- Registru *JavaScript* paketa *npm*, kod procedura koje prouzrokuju veliko CPU opterećenje [9],
- Mozilinom veb-brauzeru Fajerfoks (engl. *Firefox*) [4].

2.1 Razvoj jezika *Rust*

Programski jezik *Rust* je dizajnirao Grejdon Hor (engl. *Graydon Hoare*) koji je, u to vreme, bio zaposlen u kompaniji Mozila (engl. *Mozilla*). Hor je rad na ovom jeziku započeo 2006. godine kao svoj lični projekat, na kojem je samostalno radio naredne tri godine. Sredinom 2010. godine, u projekat se uključila i sama Mozila, koja i danas sponzoriše njegov razvoj. Pored zaposlenih Mozile, pošto

je u pitanju programski jezik otvorenog koda, svoj doprinos je dalo i preko 5000 dobrovoljaca [10].

Pre nego što se Mozila priključila projektu, *Rust* je izgledao dosta drugačije nego danas. U svojoj početnoj fazi, *Rust* je bio čist funkcionalni jezik, tj. nije imao bočne efekte; takođe, postojala je i analiza stanja tipa (engl. *typestate analysis*), koja je omogućavala proveru operacija koje se mogu izvoditi nad specifičnim tipom podataka pri kompiliranju. Za razliku od ove dve osobine, neka dizajnerska rešenja su ostala do danas, kao što je imutabilnost i kontrola pristupa memoriji. [6]

Nedugo nakon priključivanja Mozile, Grejdon Hor napušta projekat 2012ġodine. U ovom periodu *Rust* dobija svoj menadžer paketa *Cargo* zajedno sa repozitorijumom paketa `crates.io`. Proces *RFC*, inspirisan procesom *PEP* programskog jezika *Python* [12], se osniva 2014ġodine u svrhu strogog kontrolisanja novina u samom jeziku.

Prva verzija programskog jezika *Rust*, odnosno *Rust* 1.0, objavljena je 2015. godine [1]. Od tad, u *Rust* zajednici je formiran model izbacivanja novih verzija gde se one distribuiraju svakih 6 nedelja, što je dinamičniji pristup u odnosu na većinu programskih jezika gde je taj period minimalno godinu dana. Ovom odlukom se stavlja akcenat na stabilnost jezika time što će svaka nova verzija biti slična svom prethodniku, dok se kod jezika sa dugim periodom između verzija očekuju velike promene, što može da šteti kompatibilnosti.

2.2 Instalacija i korišćenje sistema *Cargo*

Ukoliko se zvanična veb-stranica programskog jezika *Rust* poseti koristeći mašinu koja ima *Windows* operativni sistem, biće ponuđene instalacione datoteke za 32-bitne i 64-bitne *Windows* sisteme.

Na *GNU/Linux* operativnim sistemima potrebno je uneti sledeću komandu u terminal:

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Potvrdu da li se instalacija uspešno izvršila uspešno može se dobiti proverom verzije *Rust* kompilatora komandom:

```
rustc --version
```

Pored samog *Rust* kompilatora, u instalaciju su uključeni i sistem *Cargo* i alat *rustup*. Alat *rustup* daje mogućnost dobavljanja nove verzije *Rust*-a sa veba, kao i mogućnost deinstalacije komandama:

```
rustup update
rustup self uninstall
```

Pored toga što je menadžer paketa, *Cargo* vrši i automatizaciju prevođenja. Kreiranje novog projekta uz pomoć ovog sistema izvršava se komandom:

```
cargo new novi_projekat
```

Komandom iznad se pravi novi direktorijum `novi_projekat` koji sadrži konfiguracionu datoteku `Cargo.toml` i direktorijum `src` koji treba da sadrži izvorne datoteke obeležene ekstenzijom *.rs* i u kojem se inicijalno nalazi datoteka *main.rs*. Takođe, sa novim direktorijumom se inicijalizuje i novi *Git* repozitorijum.

Generisana `Cargo.toml` datoteka izgleda ovako:

```
1 [package]
2 name = "novi_projekat"
3 version = "0.1.0"
4 authors = ["jovan <jdmitrovic@gmail.com>"]
5 edition = "2018"
6
7 [dependencies]
```

Listing 2.1: Inicijalna *Cargo.toml* datoteka

U prvoj liniji koda, `[package]` označava sekciju koja opisuje paket koji je napravljen. Informacije koje se ovde nalaze su dobijene iz varijabli okruženja. Posle oznake `[dependencies]` se popisuju svi paketi koji su neophodni za rad sa novim paketom, tako da ih *Cargo* može dopremiti.

Po osnovnim podešavanjima, u `main.rs` datoteci se nalazi *Hello World* program.

```
1 fn main() {
2     println!("Hello, world!");
3 }
```

Cargo prevodi projekat komandom `cargo build` a prevodi ga i pokreće `cargo run`. Korišćenjem komande `cargo check` može se proveriti da li se kôd kompilira, bez generisanja izvršne datoteke, što je korisno jer je ova opcija efikasnija od kori-

šćenja pomenute `build` komande. Kompiliranjem projekta se pravi nova putanja `target/debug`, gde će se generisati izvršne datoteke.

Kompilator ima dva profila: `dev` profil, koji se koristi za prevođenje koda tokom razvoja, i `release` profil za prevođenje završne verzije programa, spremne za isporuku. Razlika između profila ogleda se u nivou optimizacije izvršnog koda: koristeći `dev` profil, kompilator će koristiti minimalan nivo optimizacije zarad bržeg prevođenja, dok se za `release` profil projekat prevodi sa maksimalnim nivoom optimizacije zarad dobijanja najboljih performansi programa. Podrazumevano ponašanje je da se koristi `dev` profil pokretanjem komande `cargo build`, dok se prevođenje u `release` profilu izvršava komandom:

```
cargo build --release
```

2.3 Osnovne karakteristike jezika

U nastavku će biti opisane bitne karakteristike programskog jezika *Rust*, koje nisu nužno u okviru jednog stila programiranja. Zastupljeno je nekoliko programskih paradigmi:

- Imperativna,
- Objektno-orijentisana, gde se umesto klasa koriste svojstva (engl. *traits*),
- Generička, u vidu generičkih tipova,
- Funkcionalna, u vidu iteratora i zatvorenja [8],
- Konkurentna [7].

Promenljive i konstante

Promenljive se mogu definisati korišćenjem kljucne reci `let`, čime se podrazumeva da je takva promenljiva zapravo imutabilna, tjòna se ne može menjati. Imutabilnost promenljivih omogućava kompilatoru da prepozna razne vrste grešaka već u fazi kompilacije. Ipak, Rust dozvoljava i definisanje mutabilnih promenljivih koriscenjem ključnih reci `let mut`.

Pored promenljivih, *Rust* dozvoljava i definisanje konstanti upotrebom ključne reči `const`. Konstante se razlikuju od imutabilnih promeljivih po tome što se mogu

definisati u bilo kom opsegu, uključujući i globalni, i po tome što konstante samo mogu imati vrednost konstantnog izraza, ali ne i vrednost izvršavanja funkcije.

U okviru jezika dozvoljeno je i tzvšakrivanje (engl. *shadowing*). Sakrivanje je ponovno definisanje promenljivih. Za razliku od korišćenja ključne reči `mut`, prilikom sakrivanja je moguće promeniti tip promenljive koja se ponovo definiše.

Tipovi

Programski jezik *Rust* je statički tipiziran jezik, ali ne zaheva pisanje tipa uz svaku promenljivu, osim ako je to neophodno; primer je korišćenje funkcije `parse`:

```
1 fn main() {
2     let num = "10".parse().expect("Nije unet broj!");
3     println!("Unet broj je: {}", num);
4 }
```

Ukoliko se pokuša prevođenje ovog koda, *Cargo* će pokazati grešku:

```
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let num = "10".parse().expect("Nije unet broj!");
  |         ^^^ consider giving `num` a type
```

Ova greška se javlja zbog toga što funkcija `parse` prima generičke parametre, te je kompilatoru neophodna informacija kojeg je tipa promenljiva `num`.

Rust sadrži četiri vrste prostih tipova: cele brojeve, brojeve zapisane u pokretnom zarezu, karaktere i Bulove konstante *true* i *false*. Celi brojevi mogu biti označeni ili neoznačeni; označeni brojevi su predstavljeni tipovima `i8`, `i16`, `i32`, `i64` i `i128`, gde broj posle karaktera `i` predstavlja veličinu tipa u bitovima. Neoznačeni brojevi su predstavljeni analogno označenim, s tim da oni počinju karakterom `u`. Postoje i tipovi `isize` i `usize` čija veličina zavisi od arhitekture. Analogon tipovima `float` i `double` iz programskog jezika *C* su tipovi `f32` i `f64`. Tip `char` je veličine četiri bajta, gde su karakteri predstavljeni *Unicode* vrednostima.

Osnovni složeni tipovi u programskom jeziku *Rust* su torke i nizovi. Torke se predstavljaju na navođenjem liste tipova, razdvojene zarazima u okviru zagrada. Naredni primer ilustruje upotrebu torki:

```
1 fn main() {
2     let koordinate: (i32, f32, i32) = (1, 2.0, 3);
```

```
    let (x, y, z) = koordinate;

    println!("x: {}, y: {}", x, koordinate.1);
}
```

Torke mogu sadržati vrednosti različitog tipa, a može im se pristupiti ili korišćenjem novih promenljivih, ili korišćenjem `.i` sintakse, čime se pristupa elementu na i-toj poziciji, gde se elementi torke broje počevši od 0.

Nizovi predstavljaju kolekciju vrednosti istog tipa i, poput torki, fiksne su veličine. Obeležavaju se sa uglastim zagradama, unutra kojih su elementi razdvojeni zarezima.

Niske se u programskom jeziku *Rust* javljaju u dva oblika: u obliku literala i obliku niske promenljive dužine (u *Rust*-u se one zovu *string*, odnosno *String*). Literalima se veličina zna pre kompilacije i njihov sadržaj se ne može promeniti, pa se one mogu čuvati na steku, dok se niske promenljive dužine moraju čuvati na hipu.

Funkcije

Funkcije se definišu ključnom reči `fn`, navođenjem imena funkcije za kojim sledi lista parametara i njihovih tipova u zagradama, razdvojenih zarezima. Nakon liste parametara, moguće je napisati tip povratne vrednosti posle oznake `->`. Ukoliko povratni tip izostane, kompilator će ga automatski izvesti. Na kraju se nalazi telo funkcije između vitičastih zagrada. Primer funkcije koja izračunava zbir kvadrata dva broja je dat u okviru listinga 2.2.

```
fn zbir_kvadrata(x: i32, y: i32) -> i32 {
  x*x + y*y
}
```

Listing 2.2: Funkcija koja izračunava zbir kvadrata

Pored standardnog korišćenja ključne reči `return` kao oznake za povratnu vrednost funkcije, funkcija će vratiti vrednost poslednje naredbe koja se ne završava delimiterom.

Kontrola toka

Rust podržava klasične načine kontrole toka sa `if`, `while` i `for` komandama. Postoji par dodatnih mogućosti koje *Rust* takođe nudi:

- Izraz `if` se može koristiti u `let` naredbama; ali tipovi moraju biti unifikabilni,
- Ukoliko je potrebno napisati beskonačnu petlju, nije neophodno koristiti `while` naredbu sa uvek netačnim uslovom, već se može koristiti ključna reč `loop`, kao u listingu 2.3, gde se *loop* koristi u svrhe nalaženja najvećeg zajedničkog delioca dva broja,
- I `while` i `for` se mogu naći u okviru `let` naredbe, s tim da se u telu petlje mora naći `break` naredba uz koju se dopisuje povratna vrednosti kao u primeru 2.3, analogno korišćenju `return` naredbe u funkcijama,
- Iteriranje kroz kolekciju pomoću `for` petlje se može vršiti ili preko indeksa trenutnog elementa, ili pomoću `iter` funkcije. Za iteriranje od kraja kolekcije, na `iter` se nadovezuje funkcija `rev` kao u listingu 2.4.

```
fn main() {
    let mut a: u32 = 65342;
    let mut b: u32 = 23456;

    let gcd = loop {
        if b == 0 || a == b {
            break a;
        }

        if a == 0 {
            break b;
        }

        if a > b {
            a -= b;
        } else {
            b -= a;
        }
    };

    println!("Najveci zajednicki delilac a i b je {}", gcd);
}
```

Listing 2.3: Korišćenje naredbe *loop* i petlje u okviru *let* naredbe

```rust
fn main() {
    let a = [1, 2, 3];

    for elem in a.iter().rev() {
        println!("{}", elem);
    }
}
```

Listing 2.4: Korišćenje `iter` i `rev` funkcija

Vlasništvo

Koncept **vlasništva** je novina u odnosu na druge programske jezike. Ova osobina dozvoljava programskom jeziku *Rust* da funkcioniše bez sakupljača otpadaka. Za razliku od sakupljača otpadaka, sistem vlasništva ne utiče ni na koji način na izvršavanje programa, jer se postupanje prema skupu pravila vlasništva proverava prilikom prevođenja.

Sistem vlasništva se može svesti na tri pravila:

1. Svaka vrednost u programskom jeziku *Rust* ima promenljivu koja je poseduje.
2. U jednom trenutku, za svaku vrednost, postoji tačno jedan vlasnik.
3. Kada promenljiva završi svoj životni vek, tada se vrednost koju ta promenljiva poseduje automatski briše iz memorije.

Primer funkcionisanja ovog sistema se može dati uz pomoć pomenutih niski promenljive dužine, odnosno tipa `String`: ako se, kao u listingu 2.5, pokuša ulančavanje pokazivača na istu vrednost, `Rust` kompilator će prikazati sledeću grešku:

```
error[E0382]: borrow of moved value: `niska1`
 --> src/main.rs:6:32
  |
2 |     let niska1 = String::from("Hello World!");
  |         ------ move occurs because `niska1` has type `String`, which
does not implement the `Copy` trait
3 |     let niska2 = niska1;
  |                  ------ value moved here
```

```
...
6 |     println!("Prva niska: {}", niska1);
  |                                ^^^^^^ value borrowed here after move
```

Kompilator ukazuje na promenu poseda vrednosti na koju pokazuje `niska1`, čime promenljiva `niska1` gubi mogućnost manipulisanja vrednošću nad kojom je imala vlasništvo, time poštujući pravilo da vrednost pripada samo jednoj promenljivoj.

```
1 fn main() {
2     let niska1 = String::from("Hello World!");
3     let niska2 = niska1;
4 
5     println!("Druga niska: {}", niska2);
6     println!("Prva niska: {}", niska1);
7 }
```

Listing 2.5: Prenos vlasništva između promenljivih

Može se uočiti da je kopiranje promenljive `niska1` plitko. Duboko kopiranje promenljivih se odvija pomoću metoda `clone` koja je implementirana za tip `String`.

Kada se vrednost promenljive šalje kao parametar funkcije, vlasništvo te vrednosti prelazi u posed te funkcije, odnosno, promenljiva iz pozivaoca se ne može koristiti nakon pozivanja funkcije. Međutim, pozvana funkcija može *vratiti* vlasništvo nad promenljivom tako što iskoristi tu promenljivu kao povratnu vrednost: time se vlasništvo te promenljive vraća pozivaocu te funkcije. Promenljiva koja je vraćena može biti bilo koja promenljiva nad kojom ta funkcija ima vlasništvo, uključujući i promenljive koje ne potiču odatle (dakle, i sami parametri funkcije).

Pozajmljivanje

Proces davanja i preuzimanja vlasništva može da bude nepogodan u nekim slučajevima; zbog toga, postoji i opcija **pozajmljivanja** (engl. *borrowing*). Pozajmljivanje omogućava funkcijama da pozajme vrednosti promeljivih, koje se moraju vratiti vlasniku po završetku izvršavanja te funkcije, što znači da se pozajmljene promenljive mogu kasnije koristiti. Promenljive se pozajmljuju koristeći simbol `&`, što je ilustrovano na listingu 2.6, gde će izvršavanje koda ispisati: `Niska „Hello world" ima 2 reci`.

```
fn main() {
    let s = String::from("Hello world");

    println!("Niska \"{}\" ima {} reci", s, broj_reci(&s));
}

fn broj_reci(s: &String) -> usize {
    s.matches(" ").count() + 1
}
```

Listing 2.6: Pozajmljivanje vrednosti promenljivih

Na sličan način se definišu i reference na promenljive. Vrednosti pozajmljenih promenljivih se ne mogu menjati preko referenci definisanih na ovaj način; u te svrhe se koriste mutabilne reference koje se obeležavaju sa `&mut`. Za razliku od referenci u drugim programskim jezicima, može postojati samo jedna mutabilna referenca za jednu promenljivu u istom dosegu u kojem ne sme postojati ni imutabilna referenca te promenljive.

U *Rust*-u je nemoguće napraviti "zalutalu" referencu (engl. *dangling reference*), tj. referencu koja referiše na prostor u memoriji koji se već oslobodio. Prilikom prevođenja koda, kompilator će dati preporuku za korišćenje statičke reference uz pomoć sintakse `&'static`. Ukoliko bi se kôd iz listinga 2.7 pokušao prevesti, kompilator ne bi dozvolio prevođenje bez `static` oznaka. Konvencija je da imena promenljivih sa statičkim životnim vekom počinju velikim početnim slovom.

```
fn main(){
   println!("{}", num());
}

fn num() -> &'static u32 {
    static N: u32 = 1234;

    return &N;
}
```

Listing 2.7: Korišćenje promenljivih sa statičkim životnim vekom

Strukture

Definisanje struktura se vrši na uobičajen način, ali *Rust* nudi mogućnosti koje skraćuju sintaksu, kao u listingu 2.8:

```
struct Student {
    ime: String,
    indeks: String,
    email: String
}

fn kreiraj_studenta(ime: String, indeks: String) -> Student {
    Student {
        ime,
        email: format!("{}@matf.bg.ac.rs", indeks),
        indeks
    }
}

fn main() {
    let student1 = kreiraj_studenta(
        String::from("Ana"),
        String::from("mi15123")
    );
    let student2 = Student {
        indeks: String::from("mr14321"),
        ..student1
    };
}
```

Listing 2.8: Kreiranje novih struktura

Sa listinga 2.8 se može primetiti kako se uz pomoć oznake … mogu definisati nove instance strukture gde se imena polja poklapaju sa imenima parametara funkcije. U istom primeru će instanca `student2` imati isto ime kao i instanca `student1`, ali će polje `indeks` imati drugačiju vrednost.

Strukture se mogu i definisati putem torki, s tim da njihova polja neće imati imena, već će im se pristupati . notacijom. Strukture definisane na ovaj način ne moraju nužno imati polja, već im se struktura i ponašanje može postaviti putem interfejsa.

Da bi se implementirali metodi u okviru struktura, koristi se ključna reč `impl` kao u listingu 2.9.

```
impl Student {
    fn promena_mejla(&mut self, novi_email: String) {
        self.email = novi_email;
    }
}
```

Listing 2.9: Definisanje metoda

Mogu se implementirati i metodi koji ne uzimaju instancu kao argument; njima se pristupa pomoću `::` notacije.

Enumeratori

Enumeratori u programskom jeziku *Rust* imaju dodatne mogućnosti u odnosu na enumeratore u drugim programskim jezicima: svaki "tip" u enumeratoru može se posmatrati kao torka vrednosti. Ovime se, na primer, postiže ponašanje tipa `Option` koji premošćava izostanak vrednosti `null`. Tip `Option` je definisan na sledeći način:

```
enum Option<T> {
    Some(T),
    None
}
```

Listing 2.10: Definisanje enumeratora

Ono što `Option` nudi jeste mogućnost izostanka neke vrednosti, ali da taj izostanak ne može uticati na integritet programa. Pre nego što se koristi, vrednost tipa `Option` se mora otpakovati (engl. *unwrap*), odnosno, mora se utvrditi da li je vrednost generičkog tipa `T` izostala. Otpakivanje se vrši `match` naredbom kao u listingu 2.11.

```
fn main() {
    let a: Option<i8> = Some(10);

    match a {
        Some(x) => println!("Vrednost promenljive a je: {}", x),
        None => println!("Promenljiva a nema vrednost!"),
    }
```

```
8 }
```

Listing 2.11: Korišćenje `match` naredbe

Ukoliko se promenljiva ne otpakuje, kompilator će ispisati odgovarajuću grešku. Slično, greška će se ispisati i kada naredba `match` ne pokrije sve moguće vrste enumeratora, gde je moguće koristiti `_` notaciju za obeležavanje podrazumevane grane.

Glava 3

Razrada

Glava 4

Zaključak

Literatura

[1] *Announcing Rust 1.0*. on-line at: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html.

[2] Matt Asay. *Why AWS loves Rust, and how we'd like to help*. on-line at: https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/. 2020.

[3] *crosvm*. on-line at: https://opensource.google/projects/crosvm.

[4] Manish Goregaokar. *Fearless Concurrency in Firefox Quantum*. on-line at: https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html. 2017.

[5] *IoT Edge Security Daemon edgelet*. on-line at: https://github.com/Azure/iotedge/tree/master/edgelet.

[6] Steve Klabnik. *The History of Rust*. on-line at: https://dl.acm.org/doi/10.1145/2959689.2960081. 2016.

[7] Steve Klabnik i Carol Nichols. *Fearless Concurrency*. on-line at: https://doc.rust-lang.org/book/ch16-00-concurrency.html. 2019.

[8] Steve Klabnik i Carol Nichols. *Functional Language Features: Iterators and Closures*. on-line at: https://doc.rust-lang.org/book/ch13-00-functional-features.html. 2019.

[9] *Rust Case Study: Community makes Rust an easy choice for npm*. on-line at: https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf.

[10] *Rust contributors*. on-line at: https://thanks.rust-lang.org/.

[11] *Stack Overflow Developer Survey*. on-line at: https://insights.stackoverflow.com/survey/2020. 2020.

[12] Barry Warsaw i dr. *PEP 1 – PEP Purpose and Guidelines*. on-line at: https://www.python.org/dev/peps/pep-0001/. 2001.

Biografija autora

Jovan Dmitrović (*Gornji Milanovac, 17.11.1995.*)