Dr Dinu Dragan

PARALELNE I DISTRIBUIRANE ARHITEKTURE I JEZICI (ČAS 3)

ŠTA RADIMO DANAS?

- Izrazi
- Selekcije
- Petlje
- Funkcije

JEZIK IZRAZA

IZRAZ VS. ISKAZ

- Šta je to izraz?
- Šta je to iskaz?
- Zašto se te 2 stvari razliku?
- Šta je složenije?
- Kako je u C-u?

  5 * pom – 12 (ovo je?), **5 * pom – 12;** (ovo je?)
- Rust je jezik izraza (**An Expression Language**)
- Izrazi rade sav posao u Rust-u, tj. nameštено je da sve u Rustu bude izraz, čak i selekcije, petlje, funkcije, … **sve vraćaju vrednost**
- Izrazi i operatori imaju svoje prioritete izvršavanja
- Šta znači prioritet izvršavanja?

IZRAZ VS. ISKAZ

- Čisto da budemo sigurni:

```
fn main() {
    let i = {
        // This is a statement
        let j = 69;
        // This is an expression
        j + 1
    };

    println!("Hello, world! {}", i);
}
```

statement == iskaz

espression == izraz

- Deklaracije funkcija su isto iskazi (obratiti pažnju na upotrebljen pojam!)
- **let** iskaz se ne može dodeliti drugoj promenljivoj

```
// This code doesn't compile!
fn main() {
    let x = (let y = 0);
}
```

PRIORITET IZRAZA

Expression type	Example	Related traits
Array literal	`[1, 2, 3]`	
Repeat array literal	`[0; 50]`	
Tuple	`(6, "crullers")`	
Grouping	`(2 + 2)`	
Block	`{ f(); g() }`	
Control flow expressions	`if ok { f() }`	
	`if ok { 1 } else { 0 }`	
	`if let Some(x) = f() { x } else { 0 }`	
	`match x { None => 0, _ => 1 }`	
	`for v in e { f(v); }`	`std::iter::IntoIterator`
	`while ok { ok = f(); }`	
	`while let Some(x) = it.next() { f(x); }`	
	`loop { next_event(); }`	
	`break`	
	`continue`	
	`return 0`	
Macro invocation	`println!("ok")`	
Path	`std::f64::consts::PI`	
Struct literal	`Point {x: 0, y: 0}`	

PRIORITET IZRAZA

Expression type	Example	Related traits
Tuple field access	`pair.0`	`Deref`, `DerefMut`
Struct field access	`point.x`	`Deref`, `DerefMut`
Method call	`point.translate(50, 50)`	`Deref`, `DerefMut`
Function call	`stdin()`	`Fn(Arg0, ...) -> T`, `FnMut(Arg0, ...) -> T`, `FnOnce(Arg0, ...) -> T`
Index	`arr[0]`	`Index`, `IndexMut` `Deref`, `DerefMut`
Error check	`create_dir("tmp")?`	
Logical/bitwise NOT	`!ok`	`Not`
Negation	`-num`	`Neg`
Dereference	`*ptr`	`Deref`, `DerefMut`
Borrow	`&val`	
Type cast	`x as u32`	
Multiplication	`n * 2`	`Mul`
Division	`n / 2`	`Div`
Remainder (modulus)	`n % 2`	`Rem`
Addition	`n + 1`	`Add`

PRIORITET IZRAZA

Expression type	Example	Related traits
Subtraction	`n - 1`	`Sub`
Left shift	`n << 1`	`Shl`
Right shift	`n >> 1`	`Shr`
Bitwise AND	`n & 1`	`BitAnd`
Bitwise exclusive OR	`n ^ 1`	`BitXor`
Bitwise OR	`n | 1`	`BitOr`
Less than	`n < 1`	`std::cmp::PartialOrd`
Less than or equal	`n <= 1`	`std::cmp::PartialOrd`
Greater than	`n > 1`	`std::cmp::PartialOrd`
Greater than or equal	`n >= 1`	`std::cmp::PartialOrd`
Equal	`n == 1`	`std::cmp::PartialEq`
Not equal	`n != 1`	`std::cmp::PartialEq`
Logical AND	`x.ok && y.ok`	
Logical OR	`x.ok || backup.ok`	
End-exclusive range	`start .. stop`	
End-inclusive range	`start ..= stop`	

PRIORITET IZRAZA

Expression type	Example	Related traits
Assignment	`x = val`	
Compound assignment	`x *= 1`	`MulAssign`
	`x /= 1`	`DivAssign`
	`x %= 1`	`RemAssign`
	`x += 1`	`AddAssign`
	`x -= 1`	`SubAssign`
	`x <<= 1`	`ShlAssign`
	`x >>= 1`	`ShrAssign`
	`x &= 1`	`BitAndAssign`
	`x ^= 1`	`BitXorAssign`
	`x |= 1`	`BitOrAssign`
Closure	`|x, y| x + y`	

PRIORITET IZRAZA

- Svi operatori koji se mogu povezivati su levo asocirani (left-associative)
- To znači da ako imamo **a – b – c** oni se grupišu kao **(a – b) – c** , a ne kao **a – (b – c)**
- Operatori za poređenje, dodeljivanje vrednosti i operatori opsega (**..** i **..=**) se ne mogu nadovezivati

BLOKOVI I TAČKA ZAREZ

- Najopštiji izrazi su blokovi
- Blokovi rezultuju vrednostima koji se mogu umetnuti bilo gde se vrednost traži (pod uslovom da se završe izrazom a ne iskazom)

```
let display_name = match post.author() {
    Some(author) => author.name(),
    None => {
        let network_info = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
};
```

- Vrednost bloka je vrednost poslednjeg izraza u bloku (ako nema tačke zareza), ako poslednji izraz u bloku ima tačku zarez, onda je povratna vrednost bloka **( )**
- Većina izraza u Rustu se završava tačkom zarez i to ima isto značenje kao i u C/C++

BLOKOVI I TAČKA ZAREZ

- Tačka zarez ima funkciju u Rustu
- Na nekim mestima je obavezna (kod dodele vrednosti)
- Na nekim mestima kada se koristi znači da se povratna vrednost odbacuje, tj. da se ne vraća povratna vrednost
- Na nekim mestima kada se izostavi, znači da se vrednost vraća

```
let msg = {
    // let-declaration: semicolon is always required
    let dandelion_control = puffball.open();

    // expression + semicolon: method is called, return value dropped
    dandelion_control.release_all_seeds(launch_codes);

    // expression with no semicolon: method is called,
    // return value stored in `msg`
    dandelion_control.get_status()
};
```

BLOKOVI I TAČKA ZAREZ

- To što blok vraća vrednost je vrlo korisna stvar, ali nekada može dovesti do zanimljivih grešaka

```
...
if preferences.changed() {
    page.compute_size()  // oops, missing semicolon
}
...
```

- Poruka koju ćete dobiti:

```
error[E0308]: mismatched types
22 |         page.compute_size()  // oops, missing semicolon
   |         ^^^^^^^^^^^^^^^^^^^- help: try adding a semicolon: `;`
   |         |
   |         expected (), found tuple
   |
   = note: expected unit type `()`
              found tuple `(u32, u32)`
```

- Zašto je izostaviti ovde tačku zarez greška?

DEKLARACIJE

- Tipična deklaracija u Rustu ide sa **let**
- Rust dozvoljava deklaracije složenih struktura unutar bloka
- Moguće je definisati funkciju unutar funkcije (vidljiva je unutar celog bloka funkcije u okviru koje je definisana, ali ne može da pristupa lokalnim promenljivama spoljne funkcije)

```
use std::io;
use std::cmp::Ordering;

fn show_files() -> io::Result<()> {
    let mut v = vec![];
    ...

    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) -> Ordering {
        a.timestamp.cmp(&b.timestamp)   // first, compare timestamps
            .reverse()                  // newest file first
            .then(a.path.cmp(&b.path))  // compare paths to break ties
    }

    v.sort_by(cmp_by_timestamp_then_name);
    ...
}
```

DEKLARACIJE

- Rust nedozvoljava nadovezivanje dodele vrednosti, tj.:

```
fn main() {
    let x = (let y = 6);
}
```

- **let y = 6** ne vraća vrednost, tako da tu nastaje greška

```
 $ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found statement (`let`)
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^^^^^^^
  |
  = note: variable declaration using `let` is a statement

error[E0658]: `let` expressions in this position are unstable
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^^^^^^^
  |
  = note: see issue #53667 <https://github.com/rust-lang/rust/issues/53667> for more information

warning: unnecessary parentheses around assigned value
 --> src/main.rs:2:13
  |
2 |     let x = (let y = 6);
  |             ^         ^
  |
  = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
  |
2 -     let x = (let y = 6);
2 +     let x = let y = 6;
  |

For more information about this error, try `rustc --explain E0658`.
warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` due to 2 previous errors; 1 warning emitted
```

SELEKCIJE

SELEKCIJE

- Rust implementira 2 vrste selekcija
- **if else** – standardna jednostruka selekcija
- **match** – višestruka selekcija

IF SELEKCIJA

- Standardna forma:

```
if condition1 {
    block1
} else if condition2 {
    block2
} else {
    block_n
}
```

- Svaki uslov mora rezultovati **boolean** vrednošću
- Vitičasta zagrada je obavezna, ali zato nema potreba za zagradom oko uslova (šta će nam?)
- Zašto je ona u C-u obavezna?
- Naravno, **else if** blok, kao i poslednji **if** blok nisu obavezni

IF SELEKCIJA

- Primer višestruke if selekcije:

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

- Ispis

```
$ cargo run
  Compiling branches v0.1.0 (file:///projects/branches)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31s
    Running `target/debug/branches`
condition was true
```

IF SELEKCIJA + LET

- **if** selekcija se može kombinovati sa **let**

```
if let pattern = expr {
    block1
} else {
    block2
}
```

- Dati izraz, **expr**, se ili poklapa sa šablonom, **pattern**, u kom slučaju se **blok1** pokreće, ili se ne podudara, u kom slučaju se **blok2** se pokreće
- Ponekad je ovo lep način da se podaci izvuku iz **Option** ili **Result** tipova
- Primer na sledećem slajdu
- Ne razmatrati **if let** kao **if** i **let** već kao **if let** sintaksu (zasebnu konstrukciju koja se koristi u specifičnim situacijama)

IF SELEKCIJA + LET


```
if let Some(cookie) = request.session_cookie {
    return restore_session(cookie);
}

if let Err(err) = show_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    politely_accuse_user_of_being_a_robot();
} else {
    session.mark_as_human();
}
```

- Ovo se može ostvariti sa **match** upravljačkom strukturom

```
match expr {
    pattern => { block1 }
    _ => { block2 }
}
```

IF SELEKCIJA + LET

- U **match** nas zanima da odradimo nešto kada se desi poklapanje

```
let config_max = Some(3u8);
match config_max {
    Some(max) => println!("The maximum is configured to be {}", max),
    _ => (),
}
```

- Pošto se ništa ne radi sa **None** delom u match-u, moramo imati i **_ => ()**

- U **if let** možemo odraditi isto što i match bez ograničenja da gledamo alternative grane

```
let config_max = Some(3u8);
if let Some(max) = config_max {
    println!("The maximum is configured to be {}", max);
}
```

- Ovde **Some(max)** omogućuje da se **max** poveže (binds) sa vrednost u **Some**

LET NEŠTO = IF SELEKCIJA

- Svi blokovi **if** selekcije moraju da završe istom vrednošću

```
let suggested_pet =
    if with_wings { Pet::Buzzard } else { Pet::Hyena };  // ok

let favorite_number =
    if user.is_hobbit() { "eleventy-one" } else { 9 };  // error

let best_sports_team =
    if is_hockey_season() { "Predators" };  // error
```

- Zašto je poslednje greška?

MATCH

- Rust ima izuzetno moćnu konstrukciju kontrole toka izvršavanja programa koja se zove **match**
- **match** omogućuje poređenje vrednost sa serijom mogućih vrednosti iz nekog obrasca (**pattern**)
- Izvršava se kod na osnovu toga koji obrazac odgovara
- Obrasci mogu biti sastavljeni od literalnih vrednosti, imena promenljivih, džoker znakova (**wildcarda**) i mnogih drugih stvari
- Upotrebljivost ove kontrole proizilazi iz izražajnosti obrasca (**pattern expressiveness**)
- Kompajler garantuje da će sve grane biti pokrivene
- Svaka grana je zapravo izraz i vrednost tog izraza će se vratiti kao vrednost **match** strukture ako dolazi do poklapanja
- Ako je potrebno više izraza da se izvrše u grani, onda se koristi blok

MATCH

- **match** jako liči na C-ov **switch** (ali sa dodatkom **break** naredbe)

```
match value {
    pattern => expr,
    ...
}
```

- Zarez se može izostaviti ako je **expr** zapravo blok
- Svaka grana će se izvršiti tačno jednom
- Čim pronađe granu koja odgovara uslovu, ne gleda dalje
- Kompajler neće dozvoliti **match** koji ne pokriva sve moguće kombinacije (nema nedefinisanih stanja) – **Matches Are Exhaustive**

```
let score = match card.rank {
    Jack => 10,
    Queen => 10,
    Ace => 11
};  // error: nonexhaustive patterns
```

MATCH

- Primer pokrivanja svih kombinacija:

```
match code {
    0 => println!("OK"),
    1 => println!("Wires Tangled"),
    2 => println!("User Asleep"),
    _ => println!("Unrecognized Error {}", code)
}
```

- Svaka grana će se izvršiti tačno jednom
- _ predstavlja **wildcard** za sve preostale vrednosti (**default:** u **switch**-u)
- Redosled navođenja obrasca nije slučajan (zašto?)
  - što je obrazac na višoj poziciji, to je obrazac bitniji
  - da se _ nalazi ispred brojeva, nikada ne bih ušao u ostale grane, jer je u suštini _ ispunjen za bilo koju vrednost – kompajler bi ovde izbacio upozorenje

MATCH

- Sve grane **match** strukture moraju rezultovati istim tipom

```
let suggested_pet =
    match favorites.element {
        Fire => Pet::RedPanda,
        Air => Pet::Buffalo,
        Water => Pet::Orca,
        _ => None  // error: incompatible types
    };
```

- Kako znamo da se tipovi ne poklapaju?

PETLJE

PETLJE

- Rust podržava 4 vrste petlje

```
while condition {
    block
}

while let pattern = expr {
    block
}

loop {
    block
}

for pattern in iterable {
    block
}
```

- Svaka petlja je izraz, ali **for** i **while** uvek vraćaju **( )** kao rezultat
- Nema **do-while**, ali se može simulirati ili napraviti makro

WHILE I WHILE LET

- **while** petlja se ponaša isto kao **while** u C-u, uslov mora biti **boolean** vrednost (nema zagrade, ali moraju se koristiti vitičaste zagrade oko bloka, čak i ako je samo jedna linija koda)

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

- **while let** se ponaša isto kao **if let**, na početku svake iteracije, vrednost **expr** se ili poklapa sa datim obrascem, u kom slučaju se blok izvršava, ili ne, u kom slučaju se izlazi iz petlje
- Promenljiva iz obrasca se povezuje u sa promenljivom iz izraza

WHILE LET PRIMER

- **While let** se zapravo koristi u situacija kada je potrebno više puta pokupiti neku opcionu vrednost, gde postoji neki obrazac

```
// Repeatedly try this test.
loop {
    match optional {
        // If `optional` destructures, evaluate the block.
        Some(i) => {
            if i > 9 {
                println!("Greater than 9, quit!");
                optional = None;
            } else {
                println!("`i` is `{:?}`. Try again.", i);
                optional = Some(i + 1);
            }
            // ^ Requires 3 indentations!
        },
        // Quit the loop when the destructure fails:
        _ => { break; }
        // ^ Why should this be required? There must be a better way!
    }
}
```

WHILE LET PRIMER

- **While let** se zapravo koristi u situacija kada je potrebno više puta pokupiti neku opcionu vrednost, gde postoji neki obrazac

```
fn main() {
    // Make `optional` of type `Option<i32>`
    let mut optional = Some(0);

    // This reads: "while `let` destructures `optional` into
    // `Some(i)`, evaluate the block (`{}`). Else `break`.
    while let Some(i) = optional {
        if i > 9 {
            println!("Greater than 9, quit!");
            optional = None;
        } else {
            println!("`i` is `{:?}`. Try again.", i);
            optional = Some(i + 1);
        }
        // ^ Less rightward drift and doesn't require
        // explicitly handling the failing case.
    }
    // ^ `if let` had additional optional `else`/`else if`
    // clauses. `while let` does not have these.
}
```

LOOP

- **loop** petlja se izvršava beskonačno

```
fn main() {
    loop {
        println!("again!");
    }
}
```

- Control+C će vas izvući iz beskonačne petlje

```
$ cargo run
  Compiling loops v0.1.0 (file:///projects/loops)
   Finished dev [unoptimized + debuginfo] target(s) in 0.29s
    Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

- Blok u loop petlji se izvršava sve dok se ne izvrši **break**, **return**, ili nit izvršavanja ne počne da paniči

LOOP

- Za razliku od **while** i **for**, **loop** petlja se može iskoristiti za vraćanje vrednosti

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

- Vrednost se vraća, tako što se posle **break** naredbe doda vrednost koja se želi vratiti

CONTINUE I BREAK

- U radu sa petljama se koriste **continue** i **break**

```
// Read some data, one line at a time.
for line in input_lines {
    let trimmed = trim_comments_and_whitespace(line);
    if trimmed.is_empty() {
        // Jump back to the top of the loop and
        // move on to the next line of input.
        continue;
    }
    ...
}
```

- Kada ima više ugnježdenih petlji, **continue** i **break** se uvek odnose na najugnježdeniju petlju u toj tački (**innermost loop at that point**)
- Ako se želi da se **continue** i **break** odnose na određenu od ugnježdenih petlji, mogu se koristiti **loop labele**
- Labele se obeležavaju (počinju) jednim navodnikom (single quote)

  ’labela_petlje:

CONTINUE I BREAK


```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

CONTINUE I BREAK


```
 $ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.58s
     Running `target/debug/loops`
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

CONTINUE I BREAK

- **break** može da ima i labelu i povratnu vrednost

```
// Find the square root of the first perfect square
// in the series.
let sqrt = 'outer: loop {
    let n = next_number();
    for i in 1.. {
        let square = i * i;
        if square == n {
            // Found a square root.
            break 'outer i;
        }
        if square > n {
            // `n` isn't a perfect square, try the next
            break;
        }
    }
};
```

ZAŠTO LOOP?

- Rust kompajler analizira tok programa
  - Rust proverava da li svaka putanja kroz funkciju vraća vrednost očekivanog tipa; da bi ovo uradio ispravno, mora da zna da li je moguće doći do kraja funkcije
  - Rust proverava da se lokalne varijable nikada ne koriste neinicijalizovane; ovo podrazumeva proveru svake putanje kroz funkciju da bi se uverilo da ne postoji način da se dođe do mesta gde se promenljiva koristi bez prethodnog prolaska kroz kod koji je inicijalizuje
  - Rust upozorava na nedostupan kod; kod je nedostupan ako do njega ne dođe nijedan put kroz funkciju
- To su tzv. provere osetljive na tok (**flow-sensitive**)
- Rust je jednostavan, podrazumeva da je svaki uslov ili **true** ili **false** i ne proverava uslove petlje

ZAŠTO LOOP?

- Zato je ovaj kod prepoznat kao greška (iako zapravo nije)

```
fn wait_for_process(process: &mut Process) -> i32 {
    while true {
        if process.wait() {
            return process.exit_code();
        }
    }
} // error: mismatched types: expected i32, found ()
```

- Rešenje za to je **loop** koja se ponaša u smislu „reci šta hoćeš da uradiš“

FOR

- Za kretanje kroz iterativne strukture (niz, vektor i sl.) koristi se **for** upravljački izraz
- Mada se može to implementirati i sa **while**

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

- Pristup preko **while** je pak sklon greškama
- Šta nam je tu najveći problem?

FOR

- Primer sa **while** petljom bi sa **for** petljom izgledao zapravo:

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

- Zašto je ovaj kod sigurniji od **while**?
- U C-u bi kretanje kroz niz imalo sledeći oblik:

```
for (int i = 0; i < 20; i++) {
    printf("%d\n", i);
}
```

- Dok se u Rustu to isto radi:

```
for i in 0..20 {
    println!("{}", i);
}
```

FOR

- Ako se želi implementirati **for** kao brojač koristi se **..** operator koji proizvodi opseg vrednosti (**range**)
- **Range** je obična struktura sa 2 polja, početak i kraj, **0..20** je isto što i **std::ops::Range { start: 0, end: 20 }**
- Sam **range** tip je iterabilan tip, tj. implementira osobinu std::iter::IntoIterator
- Standardne kolekcije su iterabilne, kao i nizovi i isečci, pa se nad svima njima može primeti **for** petlja
- Vodite računa o Rust move sementaci! (pričaćemo još o tome)
- Ono će konzumirati vrednost
- To se rešava time što se koristi referenca na vrednost umesto same vrednosti
- Primer na sledećem slajdu

FOR

- **for** sa vrednošću

```
let strings: Vec<String> = error_messages();
for s in strings {                  // each String is moved into s here...
    println!("{}", s);
}                                   // ...and dropped here
println!("{} error(s)", strings.len()); // error: use of moved value
```

- **for** sa referencom

```
for rs in &strings {
    println!("String {:?} is at address {:p}.", *rs, rs);
}
```

- Iteracija nad **mut** referencom stvara **mut** referencu na svaki element

```
for rs in &mut strings {  // the type of rs is &mut String
    rs.push('\n');  // add a newline to each string
}
```

RETURN

- **return** izraz omogućuje izlaz iz funkcije ili bloka, vraćajući rezultujuću vrednost
- Ako je vrednost koja treba da se vrati izostavljena, **return** vraća **( )**
- Inače, **return** nije potreban za funkciju (pričaćemo o tome, malčice kasnije)

```
fn f() {     // return type omitted: defaults to ()
    return;  // return value omitted: defaults to ()
}
```

- Blok funkcije svojim poslednjim izrazom može da vrati vrednost, zato **return** nije potreban
- Ipak može se iskoristiti kako bi se napustio tekući posao (petlja ili selekcija)

RETURN

- Može se iskoristiti u situacijama kada se u jednom slučaju vraća vrednost a u drugom je potrebno iskočiti iz funkcije

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(err) => return Err(err)
};
```

- Šta se ovde dešava?

FUNKCIJA

FUNKCIJA

- Centralna tačka svih Rust programa je **main**
- Kreira se upotrebom **fn** ključne reči

```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

- Funkcija se može koristiti i pre njene definicije sve dok je u dosegu

FUNKCIJA

- Definisanje parametara funkcije i njihovo pozivanje prati standardna pravila iz drugih programskih jezika

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

- Tip parametra se mora navesti eksplicitno

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

FUNKCIJA

- Povratna vrednost funkcije se definiše pomoću parametra **->**
- Navodi se samo tip povratne vrednosti
- Povratna vrednost funkcije je jednaka vrednošću poslednjeg izraza

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {x}");
}
```

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

- Da je slučajno stavljena tačka zarez, dobila bi se greška
- Zašto?

FUNKCIJA


```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

```
$ cargo run
  Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: remove this semicolon

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` due to previous error
```