Programiranje II: poskusni izpit

17. april 2024

Čas reševanja je 60 minut. Veliko uspeha!

1. naloga (10 točk)

Za vsakega izmed spodnjih programov prikažite vse spremembe sklada in kopice, če poženemo funkcijo `main`. Za vsako spremembo označite, po kateri vrstici v kodi se zgodi.

a)

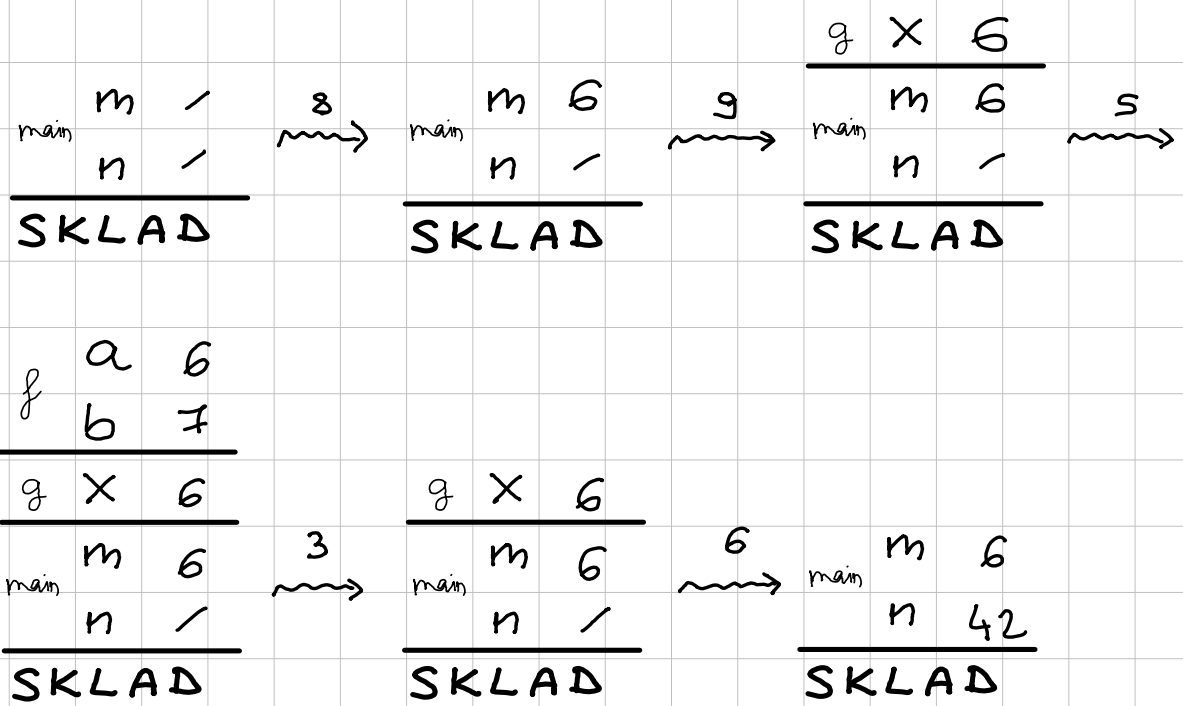
```
fn f(a: i32, b: i32) -> i32 {
  a * b
}
fn g(x: i32) -> i32 {
  f(x, x + 1)
}
fn main() {
  let m = 6;
  let n = g(m);
  println!("{n}")
}
```

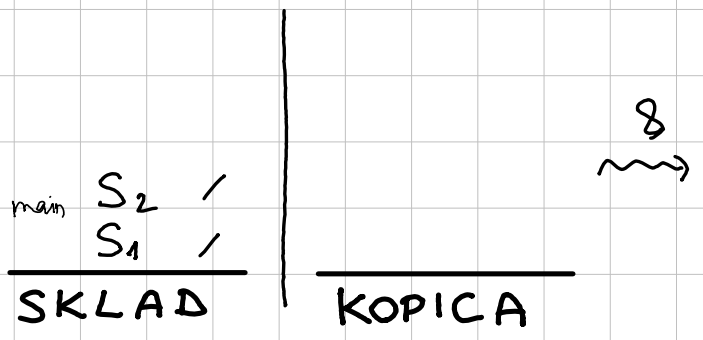
b)

```
fn f(s: String) {
  println!("{s}")
}
fn g(s: String) {
  f(s)
}
fn main() {
  let s2 = String::from("2");
  let s1 = String::from("4");
  if true {
    println!("{s2}");
  }
  g(s1);
}
```

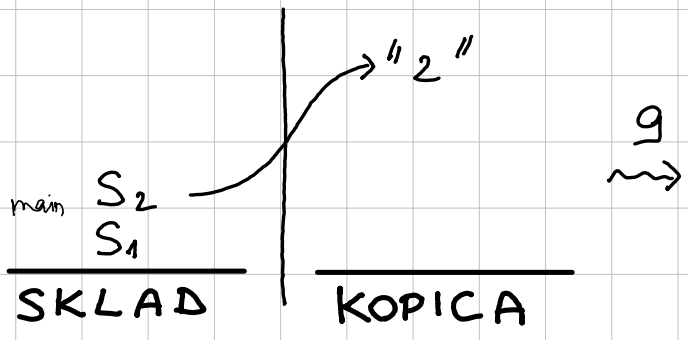
c)

```
fn f(s: &String) {
  println!("{s}")
}
fn g(s: String) {
  f(&s)
}
fn main() {
  let s1 = String::from("4");
  let s2 = String::from("2");
  g(s1);
  println!("{s2}");
}
```

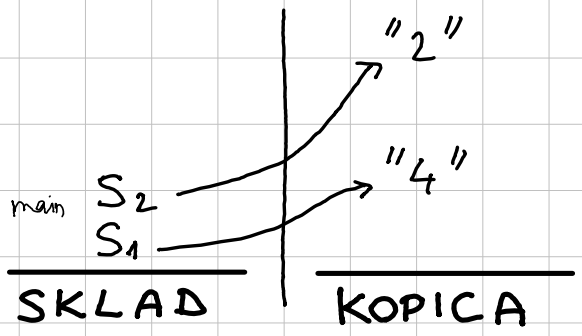

main
m /
n /
SKLAD
8
main
m 6
n /
SKLAD
9
g X 6
main
m 6
n /
SKLAD
5
f
a 6
b 7
g X 6
main
m 6
n /
SKLAD
3
g X 6
main
m 6
n /
SKLAD
6
main
m 6
n 42
SKLAD


main
S2
S1
SKLAD
KOPICA

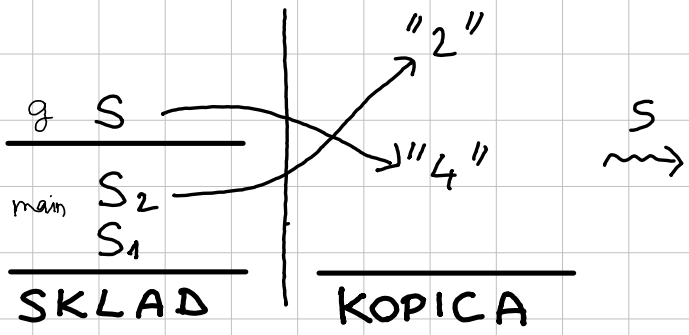
8


"2"
main
S2
S1
SKLAD
KOPICA

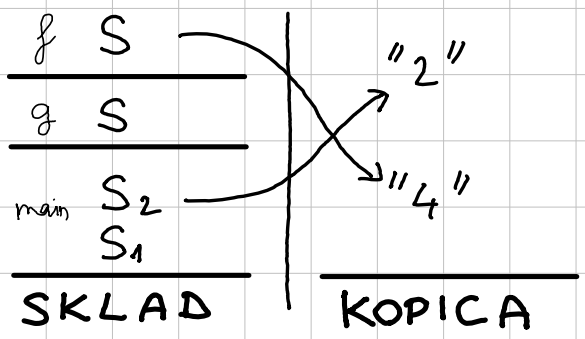
9


"2"
"4"
main
S2
S1
SKLAD
KOPICA

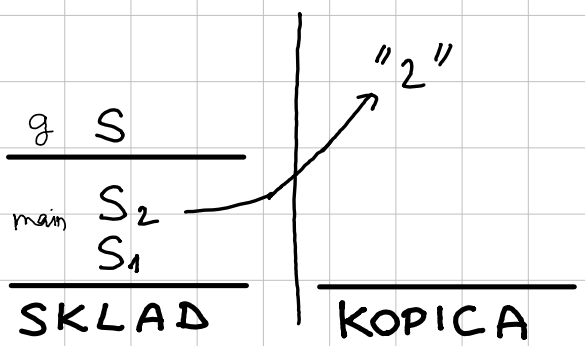
13


"2"
g S
"4"
main
S2
S1
SKLAD
KOPICA

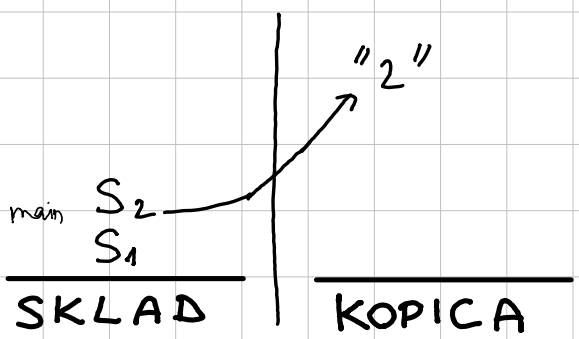
5

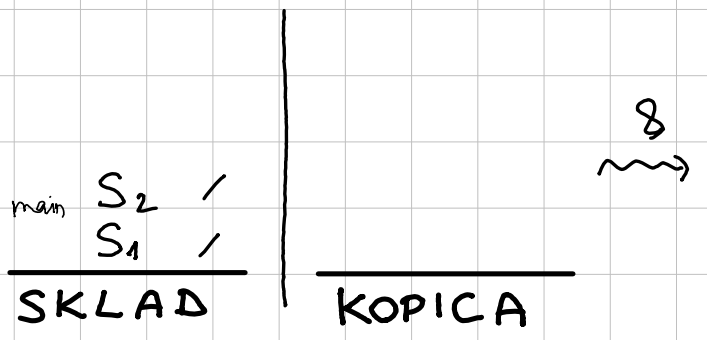

g S
"2"
g S
"4"
main
S2
S1
SKLAD
KOPICA

3

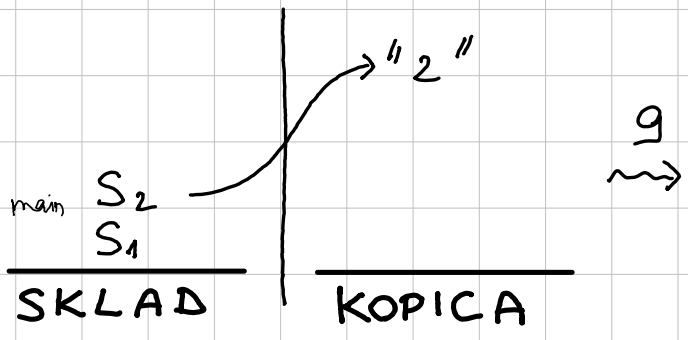

"2"
g S
main
S2
S1
SKLAD
KOPICA

6

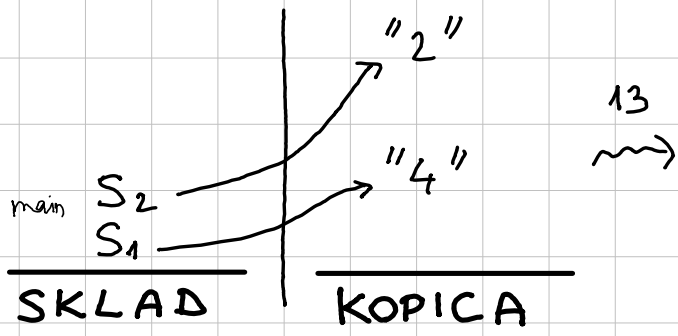

"2"
main
S2
S1
SKLAD
KOPICA


main
S2
S1
SKLAD
KOPICA

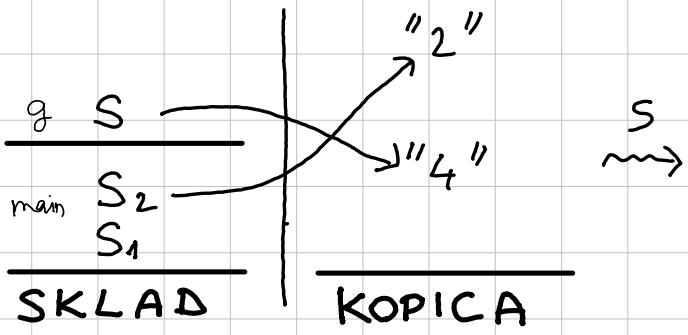
8


"2"
main
S2
S1
SKLAD
KOPICA

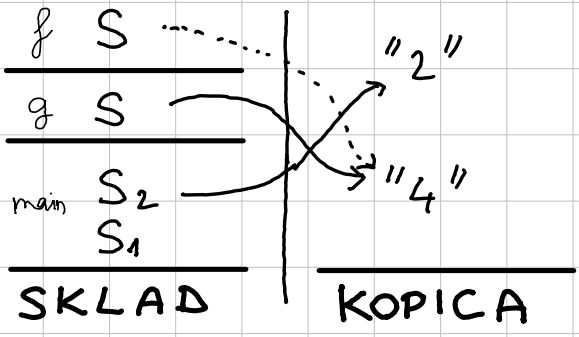
9


"2"
"4"
main
S2
S1
SKLAD
KOPICA

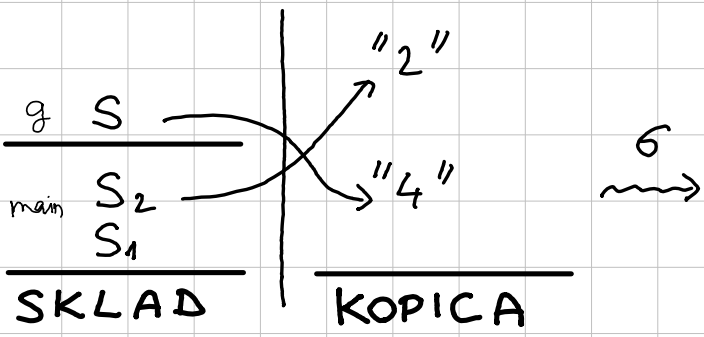
13


"2"
g S
"4"
main
S2
S1
SKLAD
KOPICA

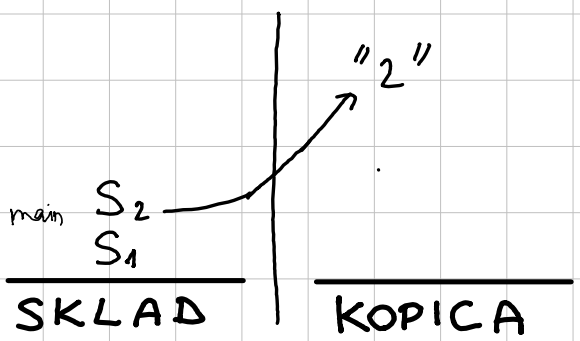
5


g S
g S
"2"
"4"
main
S2
S1
SKLAD
KOPICA

3


"2"
g S
"4"
main
S2
S1
SKLAD
KOPICA

6


"2"
main
S2
S1
SKLAD
KOPICA

2. naloga (10 točk)

Definirajmo tip množic `Set<T>`. Dopolnite signature spodnjih metod. Če v dani prostor ni treba dopisati ničesar, ga prečrtajte.

a) `fn contains(_____ self, x: ______) ________`, ki preveri, ali dana množica vsebuje element `x`.
Rešitev: `fn contains(&self, x: &T) -> bool`.

b) `fn power_set(_____ self) ________`, ki vrne potenčno množico dane množice.
Rešitev: `fn power_set(&self) -> Set<Set<T>>`.

c) `fn intersection(_____ self, other: ______) _______`, ki izračuna presek dveh množic.
Rešitev: `fn intersection(&self, other: &Self) -> Self` (namesto `Self` lahko pišete tudi `Set<T>`).

d) `fn add(_____ self, x: ______) ________`, ki v obstoječo množico doda element `x`.
Rešitev: `fn add(&mut self, x: T)`.

e) `fn into_iter(_____ self) ________`, ki iz množice naredi iterator po njenih elementih.
Rešitev: `fn into_iter(self) -> Iter<T>`.

3. naloga (30 točk)

Za vsakega izmed spodnjih programov:

1. razložite, zakaj in s kakšnim namenom Rust program zavrne;
2. program popravite tako, da bo veljaven in bo učinkovito dosegel prvotni namen.

a)

```
fn main() {
    let v = vec![1, 2, 3];
    for x in v {
      v.push(x);
    }
}
```

Rešitev: Vektor `v` je izposojen za zanko `for`, zato si ga ne moremo izposoditi še za spreminjanje za metodo `push`, saj bi spreminjanje lahko `v` prestavilo na drugo mesto v kopici, zanka `for` pa bi ostala s kazalcem na neveljaven del pomnilnika. Če želimo vektor s tako zanko podvojiti, ga moramo prekopirati z:

```
fn main() {
    let v = vec![1, 2, 3];
    for x in v.clone() {
      v.push(x);
    }
}
```

b)

```
enum Drevo {
    Prazno,
    Sestavljeno(Drevo, u32, Drevo),
}
```

Rešitev: Privzeto Rust variante v pomnilniku predstavi z oznako izbrane variante, ki ji zaporedoma sledijo predstavitve posameznih argumentov. Ob taki predstavitvi je varianta `Sestavljeno` lahko poljubno velika, zato zanjo ni mogoče rezervirati pomnilnika. Rešitev je v tem, da obe poddrevesi shranimo na kopico, zato je varianta `Sestavljeno` predstavljena samo z dvema kazalcema in vrednostjo v korenu.

```
enum Drevo {
    Prazno,
    Sestavljeno(Box<Drevo>, u32, Box<Drevo>),
}
```

c)

```
fn daljsi_niz(s1: &str, s2: &str) -> &str {
    if s1.len() > s2.len() {
        return s1;
    } else {
        return s2;
    }
}
```

Rešitev: Rust ne more določiti življenjske dobe vrnjene reference, ki bi zagotavljala dostop do rezultata ne glede na to, katerega od argumentov se izbere. Zato moramo dobe eksplicitno napisati:

```
fn daljsi_niz(s1: &'a str, s2: &'a str) -> &'a str {
  ...
}
```

d)

```
fn g(s1: &String, s2: &String) -> () {
    /// Poljubna koda, da je tip funkcije ustrezen
}

fn main() {
    let mut s1 = String::from("1");
    g(&mut s1, &s1);
}
```

Rešitev: Tako kot pri prvi podnalogi imamo hkrati spremenljivo in nespremenljivo referenco na niz `s1`. Glede na to, da funkcija g ne potrebuje spremenljivega dostopa, lahko preprosto pišemo

```
fn main() {
    let mut s1 = String::from("1");
    g(&s1, &s1);
}
```

e)

```
fn vsebuje<T>(v: &Vec<T>, x : &T) -> bool {
    for y in v {
      if x == y {
        return true
      }
    }
    return false
}
```

Rešitev: Ker primerjamo `x` in `y`, mora tip `T` podpirati vsaj značilnost `PartialEq`, saj sicer Rust nima na voljo kode, ki naj bi se izvedla ob primerjavi. Zahtevo po značilnosti moramo zapisati v tip:

```
fn vsebuje<T : PartialEq>(v: &Vec<T>, x : &T) -> bool {
  ...
}
```