Univerzitet Singidunum
www.singidunum.ac.rs

Funkcionalno programiranje

Školska 2023/24 godina

Letnji semestar

Tema 10: FP obrasci dizajna

Sadržaj

- O obrascima u programiranju
- O obrascima u JavaScript-u
- Odabrani JS obrasci
  - `Factory` funkcija
  - Kompozibilni korisnički tipovi
  - Sočiva
  - Transdjuseri

O obrascima u programiranju$_{1/2}$

- U softverskom inženjerstvu, obrazac dizajna (softvera) je uopšteno, višekratno upotrebljivo rešenje problema koji se često javlja pri razvoju softvera.
- To nije nešto što se može direktno transformisati u programski kod, već je to šablon, odnosno neka vrsta uputstva za rešavanje datog problema koji se, uz određena prilagođavanja, može koristiti u situacijama koje su slične, ali ipak različite.
- Nastali su kao rezultat prakse i predstavljaju formalizovane najbolje prakse koje programer može da koristi za rešavanje uobičajenih problema prilikom dizajniranja/izrade aplikacije ili sistema.
- Javljaju se u dva osnovna oblika: obrasci dizajna i programski obrasci

O obrascima u programiranju$_{2/2}$

- Dizajnerski obrasci se mogu posmatrati kao strukturirani pristup računarskom programiranju između nivoa programske paradigme i konkretnog algoritma.
- Shodno tome, obrasci koji će se koristiti (mogu da) zavise od programskog jezika aplikacije
  - Objektno orijentisani obrasci obično pokazuju odnose i interakcije između klasa ili objekata, bez specificiranja konačnih klasa aplikacije ili uključenih objekata.
  - Funkcionalni programski jezici insistriraju na imutabilnosti pa obrasci koji podrazumevaju promenljivo stanje mogu za te jezike biti neprikladni.
  - U jezicima koji već imaju ugrađenu podršku za rešavanje određenih klasa problema neki obrasci su nepotrebni.
  - Objektno orijentisani obrasci nisu nužno prikladni za jezike koji nisu objektno orijentisani.
- Neodgovarajuće korišćenje obrazaca u softveru može nepotrebno da uveća složenost softverskog rešenja.

O obrascima FP-a

- Funkcionalno programiranje oslanja se na dva fundamentalna obrasca programiranja:
  - Fukcija
  - Kompozicija
- Još jedan temelj obrazaca FP-a je Teorija kategorija
  - Funktor,
  - Aplikativni funktor,
  - Monada
- Pored obrazaca koji su svojstveni čistim FP jezicima, u praksi se koriste i drugi obrasci.

O obrascima u JavaScript-u

- Postoje brojne klasifikacije obrazaca dizajna softvera od kojih je najviše korišćena klasifikacija koja obrasce svrstava u sledeće grupe:
  - **Kreacioni obrasci**: bave se unapređenjem kreiranja objekata.
  - **Strukturalni obrasci**: bave se modelovanjem relacija među objektima.
  - **Obrasci ponašanja**: bave se unapređenjem komunikacije među različitim objektima.
  - **Konkurencijski obrasci**: bave se paradigmama više-nitnog proramiranja.
  - **Arhitekturalni obasci**: koriste se za opisivanje (softverske) arhitekture aplikacije
- Korišćenjem ovih bazičnih obrazaca mogu se izgraditi i složeniji obrasci.

Odabrani JS obrasci

Odabrani JS obrasci: `Factory` funkcije

- `Factory` funkcija je svaka funkcija koja nije klasa ili konstruktor a koja vraća (podrazumevano novi) objekat.
- U JavaScript-u, bilo koja funkcija može da vrati objekat. Kada to čini bez korišćenja ključne reči `new`, to je `Factory` funkcija.
- `Factory` funkcije su oduvek bile atraktivne u JavaScript-u jer nude mogućnost lakog pravljenja više instanci objekta bez zalaženja u složenost klasa i ključne reči `new`.
  - Za pravljenje više instanci objekta JS kombinuje literalnu objektni notaciju i `Factory` funkcije.

Kako izgleda `Factory` funkcija

- Factory funkcija prima svojstva budućeg objekta i vraća `this` pokazivač na kreirani objekat kao u sledećem primeru:

```
const createUser = ({ userName, avatar }) => ({
   userName,
   avatar,
   setUserName (userName) {
      this.userName = userName;
      return this;
   }
});
let mojObj1 = createUser({ userName: 'echo 1', avatar:
'echo 1.png' })
let mojObj2 = createUser({ userName: 'echo 2', avatar:
'echo 2.png' })

console.log(mojObj1);
console.log(mojObj2);
```

Kako se prave objekti pomoću `Factory` funkcije: detalji

- **Vraćanje objekata**: Pri vraćanju literalnih objekata, objekt treba da bude zatvoren u male zagrade.
- **Destrukturiranje**: Funkcija prima jedan argument (objekat) i iz njega destrukturira formalne parametre koji se zatim mogu koristiti kao varijable u dosegu tela funkcije.
- **Sračunati ključevi**: Mogu se koristiti sračunati ključevi svojstava
- **Podrazumevani argumenti**: Mogu se koristiti podrazumevani argumenti.
- **Zaključivanje tipa**: Ne postoji standardizovan način za zaključivanje tipa u JS-u što se odnosi i na `Factory` funkcije.

`Factory` detalji: vraćanje objekata

- Streličaste funkcije imaju implicitan povratak funkcije: ako se telo funkcije sastoji od jednog izraza, može se izostaviti ključna reč return:
  - Na primer, `() => 'foo'` je funkcija koja ne prima argumente i vraća string, `"foo"`.
- Treba biti pažljiv kada se vraća literalni objekat.
  - JavaScript podrazumeva da se želi kreirati telo funkcije kada se koriste zagrade, npr. `{ broken: true }`. U sledećoj liniji koda `foo:` se interpretira kao labela, a `bar` se interpretira kao izraz bez dodele ili vraćanja

```
const noop = () => { foo: 'bar' };
console.log(noop()); // undefined
```

  - Ako se želi koristiti implicitni povratak za literalni objekat, mora se literalni objekat staviti u male zagrade:

```
const createFoo = () => ({ foo: 'bar' });
console.log(createFoo()); // { foo: "bar" }
```

`Factory` detalji: destrukturiranje

- Obratite pažnju na signaturu Factory funkcije

```
const createUser = ({ userName, avatar }) => ({
})
```

- Ovde zagrade `({, })` predstavljaju destrukturiranje objekta.
  - Ova funkcija prima jedan argument (**objekat**) ali iz njega destrukturira dva formalna parametra (`userName` i `avatar`) koji se zatim mogu koristiti kao varijable u dosegu tela funkcije.
- Mogu se destrukturirati i nizovi,

```
const swap = ([first, second]) => [second, first];
console.log( swap([1, 2]) ); // [2, 1]
```

- A može se koristiti i `rest/spread` sintaksa

```
const rotate = ([first, ...rest]) => [...rest, first];
console.log( rotate([1, 2, 3]) ); // [2, 3, 1]
```

`Factory` detalji: sračunati ključevi 1/2

- Sintaksa srednje zagrade omogućuje da se dinamički odredi kom će se svojstvu pristupiti:

```
const key = 'avatar';
console.log( user[key] ); // "echo.png"
```

- Funkcija `arrToObj()` prima niz koji se sastoji od para ključ/vrednost i pretvara to u objekat korišćenjem ideje notacije srednje zagrade primenjene u kontekstu izgradnje literalnog objekta :

```
const arrToObj = ([key, value]) => ({ [key]:
value });
```

Factory detalji: sračunati ključevi 2/2

- Za željeno (moguće i izračunato) ime ključa i vrednost, funkcija `arrToObj()` kreira željeno svojstvo, odnosno objekat sa tim svojstvom :

```
let key = 'moj ' + 'avatar'
let value = 'echo.png'
let mojObj = arrToObj([key, value])
console.log ('mojObj tip: ', typeof mojObj)
console.log (mojObj)
```

- Log je:

```
mojObj tip:  object
{moj avatar: 'echo.png'}
```

`Factory` detalji: Podrazumevani argumenti$_{1/3}$

- Pri pozivu mogu da se izostave parametri sa odgovarajućim podrazumevanim vrednostima.
- Funkcija je bolje samodokumentovana jer podrazumevane vrednosti daju primere očekivanog ulaza.
- IDE i alati za statičku analizu mogu koristiti podrazumevane vrednosti da bi zaključili tip koji se očekuje za parametar.
  - Na primer, podrazumevana vrednost 1 implicira da parametar može uzeti vrednost tipa `Number`.

Factory detalji: Podrazumevani argumenti 2/3

- Primer: korišćenje podrazumevanih argumenata za implementaciju interfejsa za funkciju `createUser()`:

```
const createUser = ({
   userName = 'Anonymous',
   avatar = 'anon.png'
} = {}) => ({
   userName,
   avatar
});

console.log(
createUser({ userName: 'Some user name' }), /* {
                                                 userName: 'Some user name',
                                                 avatar: 'anon.png'} */
createUser() /* { userName: 'Anonymous', avatar: 'anon.png' } */
);
```

Factory detalji: Podrazumevani argumenti 3/3

```
const createUser = ({
    userName = 'Anonymous',
    avatar = 'anon.png'
}) => ({
    userName,
    avatar
});


console.log(
createUser({ userName: 'Some user name' }),
createUser());
/* Vratiće grešku: TypeError: Cannot read properties of
undefined (reading 'userName') at createUser
(<anonymous>:2:4) at <anonymous>:10:45 */
```

`Factory` funkcije za kompoziciju miksina

- Funkcijski miksini su `Factory` funkcije komponovane u pajplajn tako da svaka funkcija dodaje neka svojstva ili ponašanja.
- Funkcijski miksini ne zahtevaju osnovnu `Factory` funkciju ili konstruktor – jednostavno se prenosi bilo koji proizvoljni objekat u miksin koji vraća modifikovanu verziju objekta.
- `Factory` funkcije su odlične u pokretanju objekata korišćenjem lepih API-a za pozivanje. Obično su dovoljne, ali se povremeno javlja potreba za ugrađivanjem istih svojstava u različite tipove objekata.
  - U takvim situacijama prirodno je apstrahovati ta svojstva u nešto što bi omogućilo njihovo lakše ponovno korišćenje.
  - To nešto je ***funkcijski miksin***.

`Factory` funkcije za kompoziciju miksina: dodavanje konstruktora

- Ideja: miksin koji dodaje svojstvo `konstruktor` svim instancama objekta.
- Kako to uraditi: Naravno, dodavanjem svojstva `konstruktor` prototipu objekta
- Implementacija:

```
// Dodavanje konstruktora prototipu objekta
const saKonstruktorom = konstruktor => o => ({
    __proto__: { // kreiraj delegata [[Prototype]]
    konstruktor // dodaj svojstvo konstruktor u [[Prototype]]
    },
    ...o // Umešaj u novi objekat sva svojstva iz objekta o
});
```

Factory funkcije za kompoziciju miksina: Primer drona 1/5

```
// Pajplajn
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);

// Postavljanje nekih funkcijskih miksina
const saLetenjem = o => {
   let saLetenjem = false;
   return {
      ...o,
      leti () {
        saLetenjem = true;
        return this;
      },
      aterira () {
        saLetenjem = false;
        return this;
      },
      saLetenjem: () => saLetenjem
   }
};
```

`Factory` funkcije za kompoziciju miksina: Primer drona 2/5

```
// Kreiranje objekta mojDron
const kreirajDrona = () => pipe(
   saLetenjem,
   saKonstruktorom(kreirajDrona)
)({});
const mojDron = kreirajDrona();
console.log(`
konstruktor uvezan: ${ mojDron.konstruktor === kreirajDrona }
`);
console.log(`
   moze da leti: ${ mojDron.leti().saLetenjem() }
   moze da aterira: ${ mojDron.aterira().saLetenjem() }
  `);
```

- Ispis:

konstruktor uvezan: true
moze da leti: Može da leti
moze da aterira: Ne može da aterira

Factory funkcije za kompoziciju miksina: Primer drona 3/5

```
// Postavljanje još funkcijskih miksina
const saBaterijom = ({ kapacitet }) => o => {
   let pocenatPopune = 100;
   return {
     ...o,
     izvedi (procenat) {
       const ostatak = pocenatPopune - procenat;
       pocenatPopune = ostatak > 0 ? ostatak : 0;
       return this;
     },
     dobaviPopunu: () => pocenatPopune,
     dobaviKapacitet () {
       return kapacitet;
     }
   };
};
```

`Factory` funkcije za kompoziciju miksina: Primer drona 4/5

```
// Kreiranje objekta mojDron
const kreirajDrona = ({ kapacitet = '3000mAh' }) => pipe(
   saLetenjem,
   saBaterijom({ kapacitet }),
   saKonstruktorom(kreirajDrona)
)({});

const mojDron = kreirajDrona({ kapacitet: '5500mAh' });
```

`Factory` funkcije za kompoziciju miksina: Primer drona 5/5

```
console.log(`
constructor linked: ${ mojDron.konstruktor === kreirajDrona }`);
console.log(`
   moze da leti: ${ mojDron.leti().saLetenjem()}
   moze da aterira: ${ mojDron.aterira().saLetenjem()}
   baterija kapacitet: ${ mojDron.dobaviKapacitet() }
   baterija status: ${ mojDron.izvedi(50).dobaviPopunu() }%
   baterija praznjenje: ${mojDron.izvedi(75).dobaviPopunu() }%
`);
```

- Ispis:

konstruktor uvezan: true
moze da leti: Može da leti
moze da aterira: Ne može da aterira
baterija kapacitet: 5500mAh
baterija status: 50%
baterija praznjenje: 0%

Kompozibilni korisnički tipovi$_{1/6}$

- Iako programski jezici, pa i JavaScript, nude brojne ugrađene tipove, izgradnja korisničkih tipova je jedan od najčešće korišćenih obrazaca u programiranju.
- Kreiranje novog tipa je postupak kojim se kombinovanjem postojećih tipova izgrađuju novi tipovi koji su podložni proverama kao i ugrađeni tipovi.
- Kompozicija funkcija je najlakši i najčešće korišćen način za kombinovanje u JavaScript-u.
- Pokazaćmo na primeru kako se u JS-u na bazi kompozicije funkcija može izgraditi korisnički tip koji je kompozibilan.

Kompozibilni korisnički tipovi 2/6

- Posmatrajmo kod (factory funkcija koja vraća instance numeričkog tipa podataka, **t**):

```
const t = value => {
    const fn = () => value;
    fn.toString = () => `t(${ value })`;
        return fn; // Vraća se funkcja
};
const someValue = t(2); // () => value
console.log(
    someValue.toString() // "t(2)"
);
```

Kompozibilni korisnički tipovi 3/6

- Pretpostavimo da je primarni slučaj upotrebe ovako definisanog tipa sabiranje njegovih članova i razmotrimo mogućnost da sabiranje realizujemo kao kompoziciju članova tipa.
- Dakle, da vidimo da li možemo sabiranje dve vrednosti **`x`** i **`y`** da implementiramo kao **`t(x)(t(y))`**.
- Konkretno, da li možemo da isprogramiramo nešto što će da radi ovako:

```
console.log(
   t(3)(t(5)).toString() // t(8)
);
```

Kompozibilni korisnički tipovi 4/6

- Pokazuje se da se to može uraditi ako kod naše `factory` funkcije modifikujemo sledeći način:

1. Promeni se funkcija **`fn`** u funkciju **`add()`** koja vraća **`t(value + n)`** gde je **`n`** prosleđeni argument.
2. Doda se metoda **`valueOf()`** tipu **`t`** tako da nova funkcija **`add()`** može primiti instance **`t()`** kao argumente. Operator **`+`** će koristiti **`n.valueOf()`** kao drugi operand.
3. Dodele se metode funkciji **`add()`** pomoću **`Object.assign()`**.

Kompozibilni korisnički tipovi 4/6

- Naša factory funkcija sada izgleda ovako:

```
const t = value => {
    const add = n => t(value + n);

        return Object.assign(add, {
            toString: () => `t(${ value })`,
            valueOf: () => value
        });
};
```

Kompozibilni korisnički tipovi 5/6

- Sada se može primeniti kompozicija da se sračuna zbir:

```
// Komponuje funkcije sa leva na desno:
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);

// Obezbeđuje početnu vrednost za pajplajn:
const add = (...fns) => pipe(...fns)(t(0));

// Sada se komponuje/sabira
const sum = add(
   t(2),
   t(4),
   t(-1)
);

// Rezultat kompozicije je
console.log(sum.toString()) // t(2)(t(4)(t(-1))) = t(5)
```

Kompozibilni korisnički tipovi 6/6

- Ovo se može raditi sa bilo kojim tipom sve dok je operacija kompozicije smislena.
  - Na primer, za stringove to može da bude konkatenacija , za obradu digitalnih signala sumiranje signala i slično.
- U stvari, konačna odluka o korišćenju oslanja se na procenu u kolikoj meri operacija reprezentuje koncept kompozicije, odnosno koja će operacija imati najviše dobiti ako se izrazi na sledeći način:

```
const result = compose(
   value1,
   value2,
   value3
);
```

Obrazac Sočivo (`Lenses`)

- **Sočivo** je **kompozibilan par čistih geter i seter funkcija** koje se **fokusiraju na određeno polje (svojstvo) unutar objekta** i poštuju skup aksioma poznatih kao ***zakoni sočiva***.
- Sočiva tretiraju **objekat kao celinu (whole)** a **svojstvo kao deo (part)**.
- Geter prima celinu i vraća deo objekta na koji je sočivo fokusirano; ima signaturu:

```
view :: whole => part
```

- Seter prima prima celinu i vrednost za ažuriranje dela i vraća novu celinu sa ažuriranim delom; ima signaturu:

```
set :: whole => part => whole
```

Obrazac Sočivo (Lenses): primer

- Dat je niz [x, y, z] koji predstavlja koordinate neke tačke. Da bi se pribavilo ili postavilio svako polje pojedinačno, mogu se kreirati tri sočiva, po jedno za svaku osu:

```
// Geteri
const getX = ([x]) => x;
const getY = ([x, y]) => y;
const getZ = ([x, y, z]) => z;

// Seteri (u stvari, seter za y; seteri za x i z bi bili analogni)
const setY = ([x, _, z]) => y => ([x, y, z]);

console.log(
   getZ([10, 10, 100]) // 100
);
console.log(
   setY([10, 10, 10])(999) // [10, 999, 10]
);
```

Obrazac Sočivo (Lenses): Zašto?

- Zavisnosti u obliku stanja su uobičajeni način sprezanja u softveru. To je obrazac ponašanja koji omogućava objektu da promeni svoje ponašanje kada se njegovo unutrašnje stanje promeni.
- Vrlo su česte situacije kada više komponenti mogu zavisiti od nekog zajedničkog stanja, pa se, ako kasnije to stanje treba da se promeni, mora menjati logika na više mesta.
- Sočiva omogućavaju da se apstrahuje stanje iza getera i setera što omogućuje promenu oblika stanja u sočivu, bez potrebe da se menja kod kod koji zavisi od sočiva.
- Na taj način se implementira princip da mala promena u zahtevima zahteva samo malu promenu u sistemu.

Naivna implementacija sočiva 1/2

```
// Čiste funkcije za gledanje i postavljanje koje se mogu
// koristiti sa svim sočivima:
const view = (lens, store) => lens.view(store);
const set = (lens, value, store) => lens.set(value, store);

// Funkcija koja prima svojstvo (prop)i vraća naivni
// lens akcesor za to svojstvo.
const lensProp = prop => ({
   view: store => store[prop],
   // Ovo je vrlo naivno jer radi samo za objekte:
   set: (value, store) => ({
      ...store,
      [prop]: value
   })
});
```

Naivna implementacija sočiva 2/2

```
// Primer objekta skladišta:
const fooStore = {
   a: 'foo',
   b: 'bar'
};

const aLens = lensProp('a'); // lens akcesor za svojstvo a
const bLens = lensProp('b'); // lens akcesor za svojstvo b


// Destrukturiranje svojstava `a` i `b` iz sočiva korišćenjem
// funkcije `view()` .
const a = view(aLens, fooStore);
const b = view(bLens, fooStore);
console.log(a, b); // 'foo' 'bar’

// Postavljanje vrednosti u skladište korišćenjem aLens:
const bazStore = set(aLens, 'baz', fooStore);

// Uvid u novopostavljenu vrednost.
console.log( view(aLens, bazStore) ); // 'baz'
```

Obrazac Sočivo (**Lenses**): Zakoni sočiva

- Zakoni sočiva su algebarski aksiomi koji obezbeđuju da se sočivo dobro ponaša.
  1. **`view(lens, set(lens, value, store)) ≡ value`** - Ako se postavi vrednost u skladište, i odmah pogleda vrednost kroz sočivo, dobiće se vrednost koja je ažurirana (postavljena).
  2. **`set(lens, b, set(lens, a, store)) ≡ set(lens, b, store)`** - Ako se postavi vrednost sočiva na **a** a zatim odmah postavi vrednost sočiva na **b**, isto je kao da je samo postavljena vrednost na **b**.
  3. **`set(lens, view(lens, store), store) ≡ store`** - Ako se pribavi vrednost sočiva iz skladišta, a zatim odmah vrati ta vrednost u skladište, vrednost ostaje nepromenjena.
- U knjizi Erika Eliota "Composing Software" možete naći kod za verifikciju zakona sočiva za prethodnu implementaciju

Komponovanje sočiva 1/2

- Sočiva su kompozibilna.
- Komponovanje sočiva Vam je kao endoskopija u medicini: kada se komponuju sočiva, dobijeno sočivo će zaroniti duboko u objekat, obilazeći celu strukturu objekta.
- Pokazaćemo to na primeru **objekta** (`obj`) sa svojstvom (`foo`) koje je takođe **objekat** sa svojstvom (`bar`) koje je **niz**:

```
const obj = {
    foo: {
    bar: [1, 2, 3, 4]
    }
};
```

Komponovanje sočiva 2/2

```
const lensProps = [
   'foo',
   'bar',
   2 // indeks člana niza bar u objektu obj na kome je fokus
];
const lenses = lensProps.map(R.lensProp);
const truth = R.compose(...lenses);
const obj = {
   foo: {
   bar: [1, 2, 3, 4]
   }
};
console.log(
   R.view(truth, obj) // vratiće 3
);
```

Sočiva: operacija `over()`

- Mapersku funkciju `a => b` moguće je primeniti u kontekstu bilo kog funktor tipa i pokazano je da je funktorski map kompozicija.
- Na sličan način se funkcija može primeniti na vrednost fokusa u sočivu. Uobičajeno je ta vrednost istog tipa – dakle, funkcija `a => a`.
- Operacija `map` za sočiva se obično naziva „over“ u JavaScript bibliotekama.
- Funkcija `over()` omogućuje da se sadržaj u objektu postavi putem maperske funkcije.

Operacija `over()` - implementacija

```
// over = (lens, f: a => a, store) => store
const over = (lens, f, store) =>
set(lens, f(view(lens, store)), store);
```

Implementacija operacije **over()** – kompletan kod za testiranje$_{1/3}$

```
// Čiste funkcije za gledanje i postavljanje koje se mogu
// koristit sa svim sočivima:

const view = (lens, store) => lens.view(store);
const set = (lens, value, store) => lens.set(value, store);

// Funkcija koja prima svojstvo (prop)i vraća naivni
// lens akcesor za to svojstvo.

const lensProp = prop => ({
   view: store => store[prop],
   // Ovo je vrlo naivno jer radi samo za objekte:
   set: (value, store) => ({
      ...store,
      [prop]: value
   })
});
```

Implementacija operacije **over()** – kompletan kod za testiranje$_{2/3}$

```
// over = (lens, f: a => a, store) => store
const over = (lens, f, store) => set(lens, f(view(lens,
store)), store);

const uppercase = x => x.toUpperCase(); // maperska funkcija

// Primer objekta skladišta:
const fooStore = {
   a: 'foo',
   b: 'bar'
};

const aLens = lensProp('a');
const bLens = lensProp('b');
```

Implementacija operacije **over()** – kompletan kod za testiranje$_{3/3}$

```
// Sadržaj se medifikuje putem funkcije uppercase

console.log(
over(aLens, uppercase, fooStore) // { a: "FOO", b: "bar" }
);


// Seteri reflektuju zakone funktora:
{ /* ako se mapira identitetska funkcija putem over, skladište
ostaje nepromenjeno. */
const id = x => x;
const lens = aLens;
const a = over(lens, id, fooStore);
const b = fooStore;

console.log(a, b); // {a: 'foo', b: 'bar'} {a: 'foo', b: 'bar'}
}
```

Operacija **over()** – komponovanje$_{1/2}$

```
const lensProp = prop => ({
   view: store => store[prop],
   // Ovo je vrlo naivno jer radi samo za objekte:
   set: (value, store) => ({
      ...store,
      [prop]: value
   })
});

const view = (lens, store) => lens.view(store);
const set = (lens, value, store) => lens.set(value, store);

const aLens = lensProp('a');
const bLens = lensProp('b');

const over = R.curry(
(lens, f, store) => set(lens, f(view(lens, store)), store)
);
```

Operacija **over()** – komponovanje$_{2/2}$

```
{ // over(lens, f) nakon over(lens, g) je isto kao
  // over(lens, compose(f, g))
const lens = aLens;
const store = {
a: 20
};
const g = n => n + 1;
const f = n => n * 2;
const a = R.compose(
over(lens, f),
over(lens, g)
);
const b = over(lens, R.compose(f, g));
console.log(
a(store), // {a: 42}
b(store) // {a: 42}
);
}
```

Transdjuseri

Obrazac Transdjuser

- Pozadina i etimologija
  - U hardverskim sistemima za obradu signala, pretvarač (transdjuser) je uređaj koji pretvara jedan oblik energije u drugi (n.pr., audio talase u električni signal u mikrofonu). Generalnije, transformiše se jedna vrsta signala u drugu vrstu signala.
  - Potpuno analogno, transdjuser izražen u programskom kodu pretvara jedan signal u drugi signal.
- Ideja transdjusera javlja se u različitim oblastima inženjerstva oblicima koji se međusobno razlikuju do izvesne mere.
- U našem razmatranju gledaćemo ga kao **generalni koncept redjusera višeg reda — transformaciju transformacije**.

Šta rade redjuseri

```
// Sume: (1, 2) = 3
const add = (a, c) => a + c;

// Proizvodi: (2, 4) = 8
const multiply = (a, c) => a * c;

// Konkatenacija stringova: ('abc', '123') = 'abc123'
const concatString = (a, c) => a + c;

// Konkatenacija lista: ([1,2], [3,4]) = [1, 2, 3, 4]
const concatArray = (a, c) => [...a, ...c];
```

Šta je transdjuser i šta su mu odlike

- **Transdjuser** je **kompozibilni redjuser višeg reda**.
- On prima **redjuser kao ulaz** i **vraća drugi redjuser**.
- Odlike transdjusera su:
  - Transdjuseri se mogu komponovati korišćenjem jednostavne kompozicije funkcija.
  - Transdjuseri mogu da funkcionišu nad bilo kojim prebrojivim izvorom (n.pr., nizovi, stabla, strimovi, grafovi, itd....).
  - Transdjuseri su efikasni za velike kolekcije i beskonačne strimove: transdjuser vrši **enumeraciju nad signalom samo jednom**, bez obzira na broj operacija u sekvenci (pajplajnu).
  - Transdjuseri se mogu koristiti i za lenju i za pohlepnu (strogu) evaluaciju bez izmene u transdjuserskoj sekvenci (pajplajnu).

Šta je razlika između redjusera i transdjusera$_{1/2}$

- ***Redjuser*** (reducer) **prima dva ulaza** i **vraća jedan izlaz**, dakle ima sledeću signaturu:

$$f\colon (a, c) => a$$
$$g\colon (a, c) => a$$

- Šta bi u tom slučaju bilo $\boldsymbol{h}$ koje predstavlja kompoziciju redjusera $\boldsymbol{f}$ i $\boldsymbol{g}$?
  - Problem je što u ovoj kompoziciji $\boldsymbol{g}$ kao redjuser **prima dva ulaza** i **vraća samo jedan izlaz** pa bi funkcija $\boldsymbol{f}$ dobila samo jedan ulaz a ona kao redjuser **očekuje dva ulaza**.

Šta je razlika između redjusera i transdjusera$_{2/2}$

- ***Transdjuser*** (transducer) ima signaturu:

$$f\colon reducer \Rightarrow reducer$$
$$g\colon reducer \Rightarrow reducer$$
$$h\colon reducer \Rightarrow reducer$$

- Ovde sve funkcije primaju po jedan ulaz i vraćaju po jedan izlaz koji je `reducer`.
- Dakle, i broj argumenata i tipovi su saglasni (uvek je tip `reducer`) pa nema nikakvih problema sa kompozicijom.

Zašto transdjuseri 1/6

- Vrlo često je pri obradi podataka zgodno da se obrada razloži u više nezavisnih stanja koja se mogu komponovati. Na primer, vrlo je uobičajeno da se neki podaci jednom selektuju iz većeg skupa podataka i da se ti odbrani podaci dalje obrađuju.
- Na primer, da se iz zadate liste (niza) objekata

```
const friends = [
{ id: 1, name: 'Sting', nearMe: true },
{ id: 2, name: 'Radiohead', nearMe: true },
{ id: 3, name: 'NIN', nearMe: false },
{ id: 4, name: 'Echo', nearMe: true },
{ id: 5, name: 'Zeppelin', nearMe: false }
];
```

  formira niz koji sadrži vrednosti svojstva `name` svakog objekta u kome svojstvo `nearMe` ima vrednost `true`.
- Najprirodnija ideja je da se zadatak reši tako što će se prvo uraditi `filter()` da bi se podaci ekstrahovali a zatim `map()` da bi se podaci transformisali (obradili).

Zašto transdjuseri 2/6

- Za primer bi kod mogao da izgleda ovako:

```
const friends = [
{ id: 1, name: 'Sting', nearMe: true },
{ id: 2, name: 'Radiohead', nearMe: true },
{ id: 3, name: 'NIN', nearMe: false },
{ id: 4, name: 'Echo', nearMe: true },
{ id: 5, name: 'Zeppelin', nearMe: false }
];

const isNearMe = ({ nearMe }) => nearMe;
 const getName = ({ name }) => name;
 const results = friends
      .filter(isNearMe)
      .map(getName);

console.log(results); //  ["Sting", "Radiohead", "Echo"]
```

Zašto transdjuseri$_{3/6}$

- Rešenje sa prethodnog slajda je vrlo prirodno i fino će da radi za male liste, ali nosi i neke potencijalne probleme:
  - ***Tip za koji radi***: Radi samo za nizove (tip `Array`).
  - ***Performansa***: **Sintaksa dot ulančavanja za niz**, znači da JavaScript **pravi potpuno novi privremeni niz** pre nego što pređe na sledeću operaciju što zahteva vreme i memoriju.
  - ***Implementacije standardnih operacija***: Sintaksa **dot ulančavanja** zahteva **različite implementacije standardnih operacija** za tipove koji nisu tip `Array`.

Zašto transdjuseri 4/6

- Drugi način:

```
const friends = [
{ id: 1, name: 'Sting', nearMe: true },
{ id: 2, name: 'Radiohead', nearMe: true },
{ id: 3, name: 'NIN', nearMe: false },
{ id: 4, name: 'Echo', nearMe: true },
{ id: 5, name: 'Zeppelin', nearMe: false }
];

const isNearMe = ({ nearMe }) => nearMe;
const getName = ({ name }) => name;
const getFriendsNearMe = compose(
  filter(isNearMe),
  map(getName)
);
const results2 = toArray(getFriendsNearMe,friends);
  // toArray()   ga "pokreće"
console.log (results2)
```

Zašto transdjuseri 5/6

- Redjuseri mogu da mapiraju brojeve na stringove, ili objekte na nizove, ili nizove na manje nizove, ili da ne menjaju baš ništa mapirajući `{ x, y, z } -> { x, y, z }`, mogu i da filtriraju delove signala iz strima `{ x, y, z } -> { x, y }`, ili da generišu nove vrednosti za umetanje u izlazni strim, `{ x, y, z } -> { x, xx, y, yy, z, zz }`.
- Osnovni problem sa kojim se u redukciji susrećemo je što svaka faza procesiranja podataka, recimo niza, zahteva da se **obradi ceo niz** pre no što se **jedna vrednost prosledi sledećoj fazi** u pajplajnu.
  - To ograničenje znači da bi kompozicija koja koristi `Array` metode degradirala performanse jer zahteva da se kreira novi niz i da se iterira nad novom kolekcijom u svakoj fazi kompozicije.

Zašto transdjuseri 6/6

- Zamislite da imate "cev" sa dve sekcije od kojih svaka predstavlja transformaciju koja će biti primenjena na strim podataka i da imate string koji predstavlja strim.
  - Prva transformacija predstavlja filter `isEven`, a sledeća je mapiranje `double`.
- Da bi se napravila **JEDNA potpuno transformisana** vrednost iz stringa, trebalo bi **prvo propustiti CEO string kroz `filter` sekciju** i dobiti **potpuno nov filtrirani string *pre*** nego što može da se izvrši **mapiranje u `double` sekciji** .
- Kada prva vrednost konačno stigne do **`double`**, mora se **sačekati da vrednosti u celom stringu budu udvostručene** pre nego što se **može očitati bilo koji pojedinačni rezultat**.
- Dakle, ovako nešto:

```
const double = x => x * 2;
const isEven = x => x % 2 === 0;
const arr = [1, 2, 3, 4, 5, 6];
const tempResult = arr.filter(isEven);
const result = tempResult.map(double);

console.log(result); // [4, 8, 12]
```

Efikasnija alternativa$_{1/2}$

- Alternativa je da se vrednost "propušta" direktno iz filtriranog ulaza u transformaciju mapiranja, bez kreiranja novog privremenog niza u međuvremenu.
- "Propuštanje" vrednosti jednu po jednu u trenutku ima sledeće prednosti:
  - Uklanja potrebu da se iterira nad istom kolekcijom u svakoj fazi procesa transdjusinga,
  - Transdjuseri mogu da signaliziraju zaustavljanje u bilo kom trenutku, što znači da nema potrebe za dubljom enumeracijom svake faze nad kolekcijom od onoga što je potrebno da se dobile željene vrednosti.
- Postoje dva načina da se to uradi:
  - **Pull:** lenja evaluacija, ili
  - **Push:** pohlepna evaluacija.

Efikasnija alternativa$_{2/2}$

- Pull API čeka dok konzument ne zatraži sledeću vrednost.
- Push API enumerira nad izvornim vrednostima i "gura" ih kroz sekcije cevi najbrže što može.
- Transdjuserima nije važno da li se radi pull ili push.
- Transdjuseri nisu svesni strukture podataka nad kojom deluju.
- Oni prosto pozivaju redjuser koji im je prosleđen za akumuliranje nove vrednosti.

Šta su stvarno transdjuseri$_{1/3}$

- Transdjuseri su **redjuseri višeg reda**: **Redjuserske funkcije** koje **primaju redjuser** i **vraćaju novi redjuser**.
- Rič Hinkli (američki programer koji je autor FP jezika Closure i SUBP Datomic) opisuje transdjusere kao **transformacije procesa**, smatrajući da, nasuprot jednostavnoj promeni vrednosti koje teku kroz redjuser, transdjuseri menjaju procese koji deluju nad tim vrednostima.

Šta su stvarno transdjuseri$_{2/3}$

- Signature izgledaju ovako:

```
reducer = (accumulator, current) => accumulator
transducer = reducer => reducer
```

- ili detaljnije:

```
transducer = ((accumulator, current) => accumulator) =>
((accumulator, current) => accumulator)
```

Šta su stvarno transdjuseri$_{3/3}$

- Generalno govoreći, većina transdjusera treba da se primeni parcijalno na neke argumente da bi se izvršila specijalizacija. Na primer, `map` transdjuser bi mogao da izgleda ovako:

  ```
  map = transform => reducer => reducer
  ```

  ili specifičnije:

  ```
  map = (a => b) => step => reducer
  ```

- Signatura map transdjusera kaže sledeće:
  - `map` transdjuser prima funkciju mapiranja (koja se zove `transform`) i redjuser (koji se zove `step`), i vraća novi redjuser.
  - Funkcija `step` je redjuser koji će se pozvati kada se napravi nova vrednost koja će se dodati akumulatoru u sledećem koraku (`step`).

Transdjuseri: Primer za bolje razumevanje 1/5

```
const compose = (...fns) => x => fns.reduceRight((y, f) =>
  f(y), x);

const map = f => step =>
  (a, c) => step(a, f(c));

const filter = predicate => step =>
(a, c) => predicate(c) ? step(a, c) : a;

const isEven = n => n % 2 === 0;
const double = n => n * 2;

const doubleEvens = compose(filter(isEven),map(double));
const arrayConcat = (a, c) => a.concat([c]);
const xform = doubleEvens(arrayConcat);
const result = [1,2,3,4,5,6].reduce(xform, []);

console.log(result); // [4, 8, 12]
```

Transdjuseri: Primer za bolje razumevanje 2/5

- U ovom primeru `map` **primenjuje funkciju na vrednosti u nekom kontekstu**. U ovom slučaju, **kontekst je pajplajn transdjusera**. To, grubo, izgleda ovako:

```
const map = f => step =>(a, c) => step(a, f(c));
```

- Može se koristiti na sledeći način:

```
const double = x => x * 2;
const doubleMap = map(double);

const step = (a, c) => console.log(c);

doubleMap(step)(0, 4); // 8
doubleMap(step)(0, 21); // 42
```

- Nule u pozivima funkcije predstavljaju inicijalne vrednosti redjusera.
- Naravno, pretpostavka je da je `step` funkcija redjuser, ali za svrhe demonstracije može se "kidnapovati" tako da samo loguje na konzolu kao što je ovde urađeno.

Transdjuseri: Primer za bolje razumevanje 3/5

- U primeru pojednostavljeni filterski transdjuser izgleda ovako:

```
const filter = predicate => step =>
(a, c) => predicate(c) ? step(a, c) : a;
```

- Funkcija `filter` prima predikatsku funkciju (`predicate`) i propušta samo vrednosti koje zadovoljavaju predikatsku funkciju. U protivnom, vraćeni redjuser vraća neizmenjen akumulator (a).

Transdjuseri: Primer za bolje razumevanje 4/5

- Transdjuseri postaju interesantni kada se komponuju
- Kako obe ove funkcije (`filter()` i `map()`) primaju redjuser i vraćaju redjuser, mogu se komponovati jednostavnom kompozicijom funkcija:

```
const doubleEvens = compose(filter(isEven),map(double));
const isEven = n => n % 2 === 0;
const double = n => n * 2;
const compose = (...fns) => x => fns.reduceRight((y, f) =>
  f(y), x);
```

- Ovaj kod će takođe da vrati transdjuser, što znači da se mora obezbediti konačna `step()` funkcija da bi se transdjuseru kazalo kako da akumulira rezultat:

```
const arrayConcat = (a, c) => a.concat([c]);
const xform = doubleEvens(arrayConcat);
```

- Konačno, drugi argument predstavlja inicijalnu vrednost redukcije i u ovom slučaju to je prazan niz:

```
const result = [1,2,3,4,5,6].reduce(xform, []); // [4, 8, 12]
```

Transdjuseri: Primer za bolje razumevanje$_{5/5}$

- Transdjuseri pod standardnom kompozicijom funkcija (`f(g(x))`) primenjuju se od-vrha-ka-dnu/sa-leva-na-desno.
  - Dakle, korišćenje normalne kompozicije funkcija, `compose(f, g)` , znači "primeni `f` *nakon* `g`". Transdjuseri omotavaju druge transdjuser pod kompozicijom.
  - Drugim rečima, transdjuser kaže "Ja ću da uradim moj posao i *zatim* ću da pozovem sledeći transdjuser u pajplajnu ", što ima efekat "izvrtanja" steka izvršenja.

Pravila transdjusera

- Transdjuseri moraju da poštuju sledeća pravila zarad interoperabilnosti:
  - **Inicijalizacija:** Ako nije zadata inicijalna vrednost za akumulator, transdjuser mora da pozove `step` funkciju da proizvede validnu inicijalnu vrednost nad kojom će da deluje. Ta vrednost treba da predstavlja prazno stanje. Na primer, akumulator koji akumulira niz trebao bi da obezbedi prazan niz kada se njegova `step` funkcija pozove bez argumenata.
  - **Rano terminiranje:** Proces koji koristi transdjusere mora da vrši proveru i da se zaustavi kada primi redukovanu akumulatorsku vrednost. Dodatno, `step` funkcija transdjusera koji koristi ugnježdenu redukciju mora da proverava i prenosi redukovane vrednosti kada na njih naiđe.
  - **Kompletiranje (opciono):** Neki procesi transdjusinga se nikada ne kompletiraju, ali oni koji se kompletiraju trebali bi da pozovu funkciju kompletiranja da proizvede finalnu vrednost i/ili “isprazni” stanje, a transdjuseri sa stanjem bi trebali da obezbede operaciju kompletiranja koja čisti sve akumulirane resurse i potencijalno proizvodi jednu konačnu vrednost.

Transdjuseri: pravilo inicijalizacije

- Prvo treba osigurati da `map` operacija poštuje zakon inicijalizacije (prazan element).
  - Ne treba ništa specijalno da se uradi ovde već samo da se zahtev prosledi niz pajplajn korišćenjem `step` funkcije za kreiranje pretpostavljene vrednosti:

```
const map = f => step => (a = step(), c) => (
step(a, f(c))
);
```

- Deo koji je od interesa je `a = step()` u signaturi funkcije.
  - Ako nema vrednosti za `a` (akumulator), treba je kreirati tako što će se od sledećeg redjusera u lancu zahtevati da to uradi. U konačnici, to će doći do kraja pajplajna i (nadamo se) kreirati validnu inicijalnu vrednost.

Transdjuseri: rano terminiranje

- Treba signalizirati ostalim transdjuserima u pajpalajnu da je neki transdjusing završen i da ne treba da očekuju više vrednosti za obradu.
- Nakon što dobiju ovu informaciju, drugi transdjuseri mogu da odluče da zaustave dodavanje kolekciji i proces transdjusinga (kontolisan finalnom `step()` funkcijom) može da odluči da prestane sa enumeriranjem nad vrednostima.
- Proces transdjusing može da izvrši još jedan ili više poziva kao rezultat primanja redukovane vrednosti a to je poziv ***kompletiranja***. Takva namera može se signalizirati specijalnom akumulatorskom vrednošću **reduced**.

Šta je redukovana vrednost$_{1/2}$

- U najjednostavnijem slučaju to može da bude **vrednost akumulatora omotana u specijalni tip** koji se zove `reduced`. Figurativno, o ovom omotavanju može se razmišljati kao o pakovanju u kutije i označavanju tih kutija labelama (nešto kao poštanski paketi na kojima su labele “Express” ili “Fragile”).
- U osnovi, to je način da se pošalje više poruka na mestima na kojima se očekuje samo jedna vrednost. Minimalni (nestandardni) primer liftinga tipa `reduced()` može da izgleda ovako:

```
const reduced = v => ({
get isReduced () {
  return true;
},
valueOf: () => v,
toString: () => 'Reduced(${ JSON.stringify(v) })'
});
```

Šta je redukovana vrednost 2/2

- Minimalni (nestandardni) primer liftinga tipa `reduced()` može da izgleda ovako:

```
const reduced = v => ({
get isReduced () {
   return true;
},
valueOf: () => v,
toString: () => 'Reduced(${ JSON.stringify(v) })'
});
```

- Jedini striktno zahtevani delovi su:
  - **Lifting tipa:** Način da se vrednost smesti unutar tipa (n.pr., redukovana funkcija u ovom slučaju).
  - **Identifikacija tipa:** Način testiranja vrednosti da se vidi da li je to vrednost redukovanog (n.pr., `isReduced` geter) tipa.
  - **Ekstrakcija vrednosti:** Način da se vrednost "povrati" iz tipa (n.pr., **`valueOf()`**) i **`toString()`** su ovde uključeni striktno za potrebe debagovanja). To pruža mogućnost istovremene introspekcije i tipa i vrednosti na konzoli.

Kompletiranje

- “U koraku kompletiranja, transdjuser sa stanjem redukcije treba da “isprazni” stanje pre poziva funkcije kompletiranja ugnježdenog transformatora, ukoliko nije prethodno "video" redukovanu vrednost iz ugnježdenog koraka u kom slučaju treba poništiti neodlučeno stanje.”(_ Clojure transdjuserska dokumentacija).
- Drugim rečima, ako postoji više stanja za “pražnjenje” nakon što je prethodna funkcija signalizirala da je završila redukciju, korak kompletiranja je trenutak kada se time treba pozabaviti. U toj fazi može se, opciono, uraditi sledeće:
  - Poslati još jednu vrednost (“pražnjenje” nerazrešenog stanja)
  - Odbaciti nerazrešeno stanje
  - Izvšiti zahtevano “pražnjenje” stanja

Transdjusing$_{1/3}$

- Transdjusing može da radi nad različitim tipovima podataka, ali se proces može generalizovati:

```
// ovaj deo radi kuriranje (može se uzeti i iz biblioteke):
const curry = (
f, arr = []
) => (...args) => (
a => a.length === f.length ?
f(...a) :
curry(f, a)
)([...arr, ...args]);

const transduce = curry((step, initial, xform, foldable) =>
foldable.reduce(xform(step), initial)
);
```

Transdjusing 2/3

- Funkcija `transduce()` prima funkciju `step` (finalni korak u pajplajnu transdjusera), inicijalnu vrednost za akumulator, `transducer`, i parametar `foldable`.
  - Ovde je `foldable` bilo koji objekat koji obezbeđuje `.reduce()` metodu.
- Kada je definisan `transduce()`, može se lako kreirati funkcija koja vrši transdjusing u niz.
  - Prvo je potreban redjuser koji redukuje na niz:

```
const concatArray = (a, c) => a.concat([c]);
```

- Sada se može iskoristiti kurirani `transduce()` da se kreira parcijalna aplikacija koja vrši transdjusing na nizove:

```
const toArray = transduce(concatArray, []);
```

Transdjusing$_{3/3}$

- Kada postoji `toArray()` , dve linije koda mogu se zamenuti jednom i ponovo iskoristiti u puno drugih situacija:
- *// Ručni transduce:*

```
const xform = doubleEvens(arrayConcat);
const result = [1,2,3,4,5,6].reduce(xform, []);
// => [4, 8, 12]
```

- *// Automatski transduce:*

```
const result2 = toArray(doubleEvens,
[1,2,3,4,5,6]);
console.log(result2); // [4, 8, 12]
```

Transdjuser protokol$_{1/2}$

- Transdjuseri, u stvari, nisu pojedinačne funkcije. Oni se sastoje iz tri različite funkcije.
- JavaScript transdjuseri su funkcije koje primaju transdjuser i vraćaju transdjuser. Transdjuser je objekat sa tri metode:
  - **init -** Vraća validnu inicijalnu vrednost za akumulator (obično samo poziv sledećeg `step()`).
  - **step -** Primenjuje transformaciju, n.pr., za `map(f)`: `step(accumulator, f(current))`.
  - **result** - Ako je transdjuser pozvan bez nove vrednosti, ova metoda bi trebala da odredi njegov korak kompletiranja (obično `step(a)`, osim ako je transdjuser bez stanja).

Transdjuser protokol$_{2/2}$

- Evo malo manje naivne implmentacije `map` transdjusera:

```
const map = f => next => transducer({
init: () => next.init(),
result: a => next.result(a),
step: (a, c) => next.step(a, f(c))
});
```

- Po pretpostavci, većina transdjusera bi trebala da prosledi `init()` poziv sledećem transdjuseru u pajplajnu, zato što se ne zna tip transportovanih podataka pa treba napraviti validnu inicijalnu vrednost za njih.
- Dodatno, specijalni redukovani objekat koristi ta svojstva (sa imenskim prostorom @@transducer/<name>) u transdjuserskom protokolu:
  - **reduced -** Bulovska vrednost koja je uvek `true` za redukovane vrednosti.
  - **value -** Redukovana vrednost.

Primer (James Sinclair)

- Dati su podaci iz rečnika Viktorijanskog slenga kao niz (`victorianSlang`) objekata sa obeležjima `term` (termin), `found` (indikator prisustva u Google books; vrednosti `true` ili `false`) i `popularity` (skor za popularnost, numerik). Primer objekta iz niza:

```
{
    term: 'doing the bear',
    found: true,
    popularity: 108,
},
```

- Treba naći prosečnu vrednost skora popularnosti za sve stavke prisutne u Google books.
- Napomena: Cela "baza" je dole u Notes.

Prvo rešenje 1/2

```
function isFound(item) {
    return item.found;
};

function getPopularity(item) {
    return item.popularity;
}

function addScores({totalPopularity, itemCount},
popularity) {
    return {
      totalPopularity: totalPopularity + popularity,
        itemCount: itemCount + 1,
    };
}
```

Prvo rešenje 2/2

```
const initialInfo = {totalPopularity: 0, itemCount: 0};
const popularityInfo = victorianSlang
    .filter(isFound)
    .map(getPopularity)
    .reduce(addScores, initialInfo);

// Sračunavanje i ispis proseka.
const {totalPopularity, itemCount} = popularityInfo;
const averagePopularity = totalPopularity / itemCount;
console.log("Prosečna popularnost:", averagePopularity);
```

Drugo rešenje 1/2

```
function isFound(item) {
    return item.found;
};

function getPopularity(item) {
    return item.popularity;
}

function filterFoundReducer(foundItems, item) {
    return isFound(item) ? foundItems.concat([item]) : foundItems;
}

function mapPopularityReducer(scores, item) {
    return scores.concat([getPopularity(item)]);
}

function addScores({totalPopularity, itemCount}, popularity) {
    return {
        totalPopularity: totalPopularity + popularity,
        itemCount:       itemCount + 1,
    };
}
```

Drugo rešenje 2/2

```
const initialInfo    = {totalPopularity: 0, itemCount:
0};
const popularityInfo = victorianSlang
    .reduce(filterFoundReducer, [])
    .reduce(mapPopularityReducer, [])
    .reduce(addScores, initialInfo);

// Sračunavanje i ispis proseka.
const {totalPopularity, itemCount} = popularityInfo;
const averagePopularity = totalPopularity / itemCount;
console.log("Prosečna popularnost:", averagePopularity);
```

Drugo rešenje – malo revidirano

```
function makeFilterReducer(predicate) {
    return (acc, item) => predicate(item) ?
                          acc.concat([item]) : acc;
}

function makeMapReducer(fn) {
    return (acc, item) => acc.concat([fn(item)]);
}


const filterFoundReducer   =
makeFilterReducer(isFound);
const mapPopularityReducer =
makeMapReducer(getPopularity);
```

Kako to uraditi sa transdjuserom

- Transdjuseri su alat koji **modifikuje redjusere**.
- Transdjuser je funkcija koja **prima redjuser** i **vraća drugi redjuser**.
- Šta to, u stvari, znači: Transdjuser **prima redjusersku funkciju** kao ulaz i **na neki način tu funkciju transformiše** – damo mu **jedan redjuser** i on nam vrati **drugi redjuser**.
- A u našem kodu mi sada imamo tri redjusera koje bismo mogli da kombinujemo

Treće rešenje – ideja sa transdjuserom

- Mogli bismo, na primer, da primimo redjuser i da ga modifikujemo tako da se neke stavke isfiltriraju.
  - Originalni redjuser još uvek radi, ali neke vrednosti nikada ne vidi.
- Ili bismo mogli da modifikujemo redjuser tako da je svaka stavka koja mu se prosledi transformisana ili mapirana na drugu vrednost.
  - To znači da je svaka stavka transformisana pre nego što je originalni redjuser vidi.
- To može da izgleda ovako:

Treće rešenje – ideja sa transdjuserom

```
// Funkcija koja prima reducer i vraća
// novi reducer koji filtrira neke stavke tako da
// ih originalni reducer nikada ne vidi.
function makeFilterTransducer(predicate) {
  return nextReducer => (acc, item) =>
    predicate(item) ? nextReducer(acc, item) : acc;
}

// Funkcija koja prima reducer i vraća novi
// reducer koji transformiše svaki put pre
// nego što originalni reducer stigne da vidi.
function makeMapTransducer(fn) {
   return nextReducer => (acc, item) =>
          nextReducer(acc, fn(item));
}
```

Treće rešenje – kako ide računanje proseka?

```
const foundFilterTransducer = makeFilterTransducer(isFound);
const scoreMappingTransducer =
makeMapTransducer(getPopularity);

const allInOneReducer =
foundFilterTransducer(scoreMappingTransducer(addScores));

const initialInfo    = {totalPopularity: 0, itemCount: 0};
const popularityInfo =
victorianSlang.reduce(allInOneReducer, initialInfo);

// Sračunavanje i ispis proseka.
const {totalPopularity, itemCount} = popularityInfo;
const averagePopularity = totalPopularity / itemCount;
console.log("Average popularity:", averagePopularity);
```

Četvrto rešenje: nemojte sami da pravite veš-mašinu

- U poslednjem primeru smo sami napravili transdjuser.
- Naravno, prilično naivno – koliko da razumemo suštinu.
- Implementacija koja je upotrebljiva za produkciju mora da bude znatno ozbiljnija.
- Treba li sami da je pravimo? Naravno da ne treba – postoje brojne biblioteke sa implementiranim transdjuserom.
- U sledećem primeru pokazujemo kako se to radi sa bibliotekom **ramda** (https://ramdajs.com/)

Korišćenje Ramda transdjusera

```
/* Naša "baza podataka" victorianSlang i pomoćne funkcije isFound(),
getPopularity(),addScores() */

/* Uspostavljanje našeg 'transducer-a' i postavljanje naše inicijalne
vrednosti. */
const filterAndExtract = R.compose(R.filter(isFound),
                         R.map(getPopularity));
const initVal = {totalPopularity: 0, itemCount: 0};

// Ovde se dešava 'magija'.
const {totalPopularity, itemCount} = R.transduce(
    filterAndExtract, // Transducer-ska funkcija  (Ramda je 'magijom'
                      // konvertuje)
    addScores,        // Finalni reducer
    initVal,          // Inicijalna vrednost
    victorianSlang    // Strim koji želimo da obradimo
);

// I na kraju samo izbacimo prosek.
const averagePopularity = totalPopularity / itemCount;
console.log("Prosečna popularnost:", averagePopularity);
```

Zaključak

- **Transdjuseri** su **kompozibilni ridjuseri višeg reda** koji mogu da vrše redukciju nad osnovnim podacima proizvoljnog tipa.
- **Transdjuseri** proizvode kod koji **može da bude za više redova veličine efikasniji** od dot sintakse ulančavanja sa nizovima i **da rukuje sa potencijalno beskonačnim skupovima podataka bez kreiranja posredničkih agregacija** (n.pr. pomoćnih nizova).
- **Napomena:** Transdjuseri nisu *uvek* brži od ugrađenih `Array` metoda.
  - Korisni su kada se radi o velikim skupovima podataka ili dugim pajplajnovim.
  - Ako je performansa bitna, valja uraditi profilisanje (videti gde šta primenjivati da bi se postigla optimalna performansa).

Literatura za predavanje

1. Z. Konjović, Funkcionalno programiranje, **Tema 10 – FP obrasci dizajna**, slajdovi sa predavanja, dostupni na folderu **FP kurs 2023-24→Files →Slajdovi sa predavanja**
2. E. Elliot, **Composing Software - An Exploration of Functional Programming and Object Composition in JavaScript**, Leanpub, 2019.