Univerzitet Singidunum
www.singidunum.ac.rs

Funkcionalno programiranje

Školska 2023/24 godina

Letnji semestar

Tema 4: Osnove funkcionalnog programiranja

Sadržaj

- Sistem tipova i Hindley-Milner signatura tipa
- Apstrakcija i kompozicija u softveru
- Paradigme programiranja
- Funkcionalno programiranje (FP) i osnovni koncepti FP-a
  - Šta je FP
  - Osnovni koncepti FP-a
    - Funkcija
    - Nepromenljivost (imutabilnost)
    - Komponovanje
    - Deklarativno izražavanje
- Prednosti i nedostaci FP-a

Sistem tipova i Hindley-Milner signatura tipa

Sistem tipova

- U matematici, logici i računarskoj nauci **sistem tipova** je formalni sistem u kome je svakom terminu dodeljen "tip" koji definiše njegovo **značenje** i **operacije** koje se nad njim **mogu izvršiti**.
- U praktičnom računarstvu - programiranju - tipovi su meta-jezik koji pomaže da se izbegnu greške u programima.
- U praksi se koriste različiti sistemi tipiziranja od kojih je (posebno u jezicima FP-a) vrlo zastupljen sistem koji se zove "Hindley-Milner" i koji ćemo sada da objasnimo.

Signature

- Funkcije imaju ***signature*** koje se sastoje od:
  - Neobaveznog imena funkcije.
  - Liste tipova parametara koji mogu a ne moraju da budu imenovani.
  - Tipa povratne vrednosti.
- Signature tipa olakšavaju ljudima čitanje koda (puno se vidi iz signature funkcije pa sama funkcija ne mora ni da se čita) i mašinama da automatski zaključuju tipove u fazi izvršavanja.
- U JavaScript-u nije obavezna specifikacija signature tipa.
  - Međutim, JavaScript endžin određuje tipove u fazi izvršavanja programa. Ako se obezbedi dovoljno indikatora, signaturu mogu da zaključe razvojni alati kao što su integrisana razvojna okruženja i analizatori koda koristeći analizu toka podataka. Na taj način doprinosi se da kod bude čitljiviji i korektniji.

Notacija signatura

- JavaScript nema sopstvenu notaciju signatura funkcije, ali ima nekoliko "standarda" koji su u opticaju: JSDoc (istorijski značajan ali se ne koristi mnogo), TypeScript i Flow kao najveći konkurenti trenutno.
- Ima i drugih stvari poput Rtype koji preporučuje Erik Eliot za dokumentovanje (https://github.com/ericelliott/rtype)
- Prosto, nema standardizovane notacije za JavaScript pa se koriste različite stvari.
- Haskel, koji se smatra referentnim jezikom funkcionalnog programiranja koristi kurirane Hindley-Milner tipove (http://web.cs.wpi.edu/~cs4536/c12/milner-damas_principal_types.pdf) pa ćemo i mi ovde ukratko objasniti tu notaciju.

Hindley-Milner sistem signatura tipa

- Osnove
- Signatura kao osnova za rezonovanje o tipovima i njihovim implikacijama

Hindley-Milner (HM) sistem tipa: osnove

- Sistem je vrlo jednostavan, brzo se može objasniti i ne zahteva previše prakse da bi se do prihvatljivog nivoa savladao taj mali jezik.
- Evo primera:

```
// capitalize :: String -> String
const capitalize = s => toUpperCase(head(s)) +
toLowerCase(tail(s));
capitalize('smurF'); // 'Smurf'
```

Hindley-Milner (HM) sistem tipa: osnove

- U HM sistemu, funkcije se pišu kao `a -> b` gde su a i b varijable za bilo koji tip:

```
// capitalize :: String -> String
```

- Naravno, ovde ne ulazimo u implementaciju same funkcije, jedino što nas interesuje (a i samo govori puno o implementaciji funkcije) jeste signatura koja kaže sledeće: to je funkcija koja prima tip **`String`** i vraća tip **`String`**

Hindley-Milner (HM) sistem tipa: primeri$_{1/2}$

```
// strLength :: String -> Number
const strLength = s => s.length;

// join :: String -> [String] -> String
const join = curry((what, xs) => xs.join(what));

// match :: Regex -> String -> [String]
const match = curry((reg, s) => s.match(reg));

// replace :: Regex -> String -> String -> String
const replace = curry((reg, sub, s) =>
  s.replace(reg, sub));
```

Hindley-Milner (HM) sistem tipa: primeri$_{2/2}$

```
// match :: Regex -> String -> [String]
const match = curry((reg, s) => s.match(reg));
```

- Može se napisati i ovako:

```
// match :: Regex -> (String -> [String])
const match = curry((reg, s) => s.match(reg));
```

- Šta nam kaže novi zapis?
- Evo i primera primene:

```
// match :: Regex -> (String -> [String])
// onHoliday :: String -> [String]
const onHoliday = match(/holiday/ig);
```

Rezonovanje o tipovima i njihovim implikacijama

- Rezonovanje o tipovima i njihovim implikacijama ilustrovaćemo na primerima

```
/* id :: a -> a – prima vrednost tipa a i
  vraća vrednost istog tipa a */
const id = x => x;


/* map :: (a -> b) -> [a] -> [b] – prima
  funkciju koja mapira vrednost proizvoljnog
  tipa a na vrednost proizvoljnog tipa b (može
  i tip a) i niz vrednosti tipa a; vraća niz
  vrednosti tipa b */
const map = curry((f, xs) => xs.map(f));
```

Primer: glava liste, rep liste

```
//  head :: [a] -> a
var head = function(xs) {
  return xs[0];
};
console.log(head([1,2,3])) // => 1

//  tail :: [a] -> a
var tail = function(xs) {
  let last = xs.length-1
  return xs[last];
};
console.log(tail([1,2,3])) // => 3
```

Signatura kaže: Prima **niz** vrednosti **tipa a** i vraća vrednost **tipa a**

Kod dalje kaže šta stvarno radi: vraća prvi element ili poslednji element liste, ali to nama nije važno - mogla je da vrati I 0 ili 2 (indekse tih elemenata liste) a signatura ostaje ista.

Mapiranje

- Mapiranje je transformacija jedne vrednosti u drugu vrednost.
  - Na primer, ako uzmete broj 2 i pomnožite ga sa 3, vi ste broj 2 mapirali na 6 i to tako što ste primenuli funkciju koja svoj ulaz množi sa 3 .
- Pri tome, o mapiranju ne govorimo kao mutaciji ulazne vrednosti ili ponovnoj dodeli, već kao o *projektovanju* **vrednosti** iz **jedne lokacije** u **novu** vrednost **na drugoj lokaciji** – znači, poštovanje **FP principa imutabilnosti**.
- Dakle:

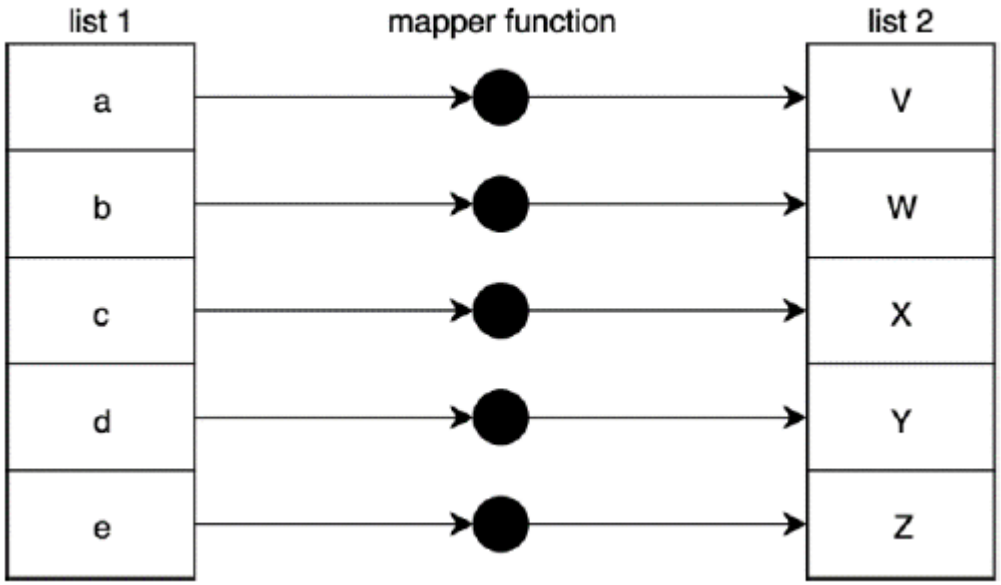
```
var x = 2, y;
y = x * 3;//ovo je ispravno (transformacija/projekcija)
x = x * 3; //ovo nije ispravno (mutacija, ponovna dodela)
```

- Odnosno, ovo je dobro:

```
var multipleBy3 = v => v * 3;
var x = 2, y;
// transformacija / projekcija
y = multiplyBy3( x );
```

`map` nad listom

- Možemo prirodno da proširimo mapiranje jedne vrednosti na mapiranje kolekcije vrednosti: `map()` nad listom je operacija koja transformiše **sve vrednosti iz liste** i projektuje (smešta) **nove vrednosti** u **novu listu**.



mapSimple (): naivna implementacija map-a

```
const mapSimple = (array,fn) => {
  let results = []
        for(const value of array)
          results.push(fn(value))
   return results;
}
// pozivi
let inList = [1,2,3]
console.log (' Ulazna lista pre poziva map: ' + inList);  // [1,2,3]
let outList = mapSimple(inList, (x) => x + 2);
console.log (' Ulazna lista nakon poziva map: ' + inList);  // [1,2,3]
console.log ( ' Izlazna lista nakon map: ' + outList); // [3,4,5]
```

Mapiranje: signatura

```
// map :: (a -> b) -> [a] -> [b] -
const map = curry((f, xs) =>
xs.map(f));
```

Signatura kaže: prima **funkciju** koja mapira vrednost proizvoljnog tipa **a** na vrednost proizvoljnog tipa **b** (može i tip **a** a ne mora) i niz vrednosti tipa a; vraća **niz** vrednosti tipa b

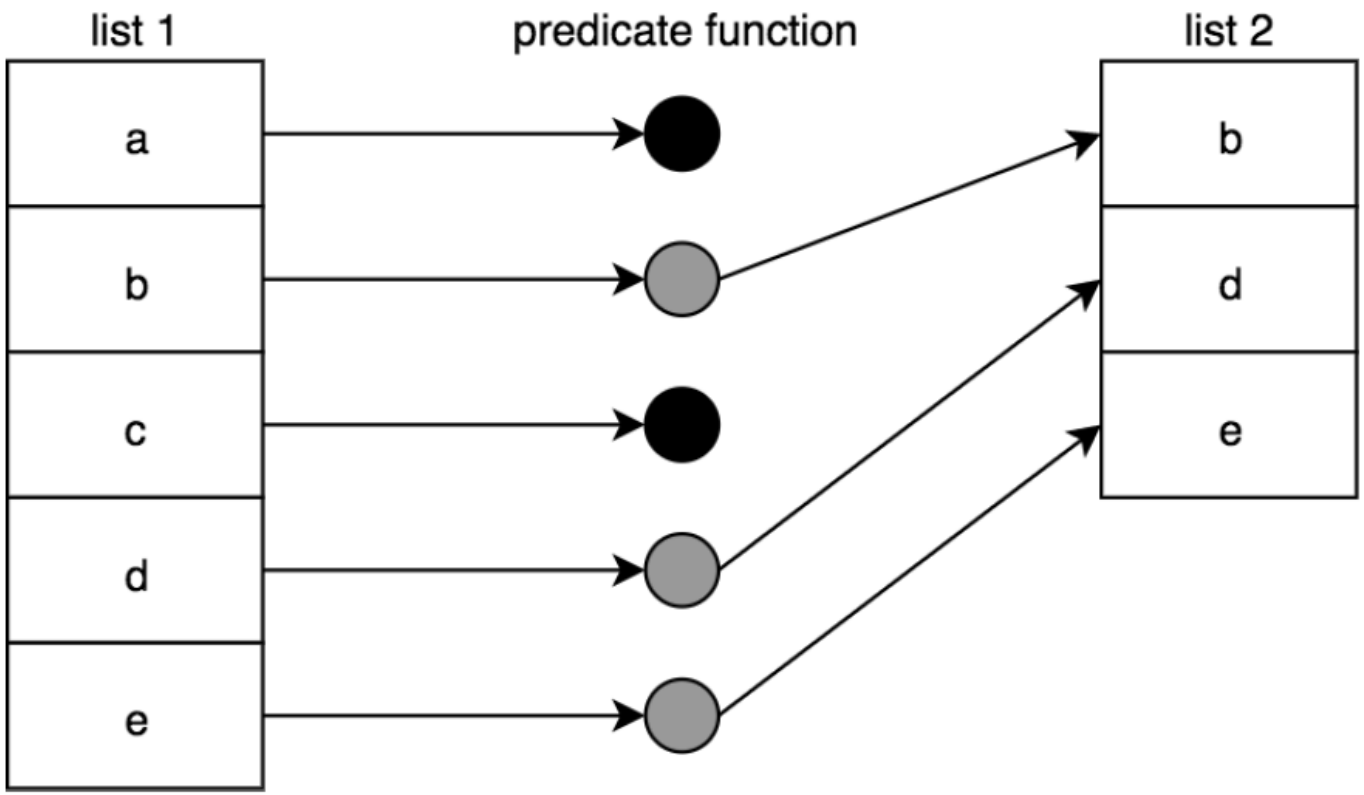
Filtriranje

- Operacija filtriranja nad listom (`filter()`) primenjuje funkciju da bi odlučila da li će svaka vrednost u originalnom nizu biti u novom nizu ili neće.
- Ta funkcija treba da vrati vrednost `true` ako vrednost treba da se uključi, ili vrednost `false` ako vrednost treba da se preskoči.
- Funkcija koja vraća vrednost `true`/`false` za takvu vrstu odlučivanja zove se ***predikatska funkcija***.

Predikatska funkcija

- Ako razmišljate o vrednosti `true` kao o indikatoru pozitivnog signala, definicija funkcije `filter()` je da Vi kažete "zadrži" vrednost (za filtriranje **u**), a ne "odbaci" vrednost (za filtriranje **iz**).
- Da bi se operacija `filter()` koristila kao isključujuća akcija, potrebno je da razmišljate o pozitivnom signalu kao indikaciji isključenja koja vraća `false` i pasivnom propuštanju vrednosti vraćanjem `true`.
- Važno je da ovo bude jasno zbog načina na koji ćete želeti da imenujete funkciju koja se koristi kao `predicateFn()`, i zbog značenja za čitljivost koda.

Filtriranje liste vrednosti



Funkcija `filter()`: naivna implementacija

```
const filter = (array,fn) => {
 let results = []
     for(const value of array)
       (fn(value)) ? results.push(value)
                   : undefined
        return results;
}
```

Filtriranje: signatura

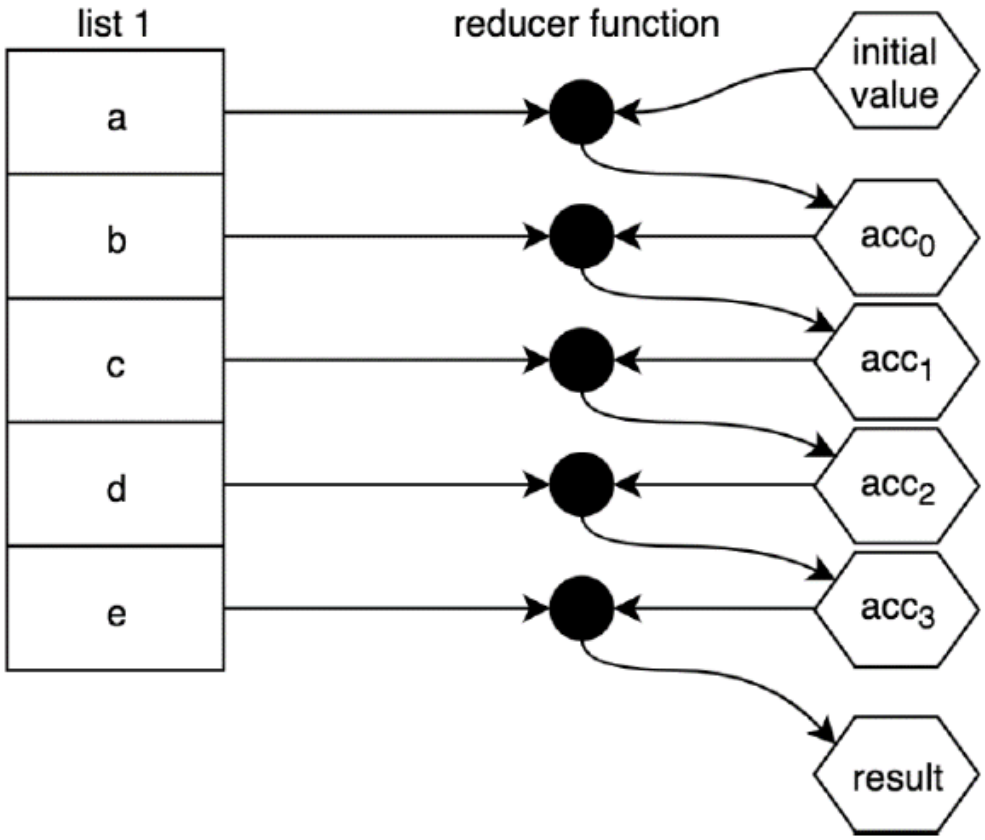
```
//  filter :: (a -> Bool) -> [a] -> [a]
var filter = curry(function(f, xs) {
  return xs.filter(f);
});
```

Signatura kaže: prima dva argumenta: pvi je funkcija koja prima tip **a** i vraća tip **Bool**, drugi je **niz** vrednosti tipa **a**; vraća **niz** vrednosti tipa **a**.

Redukcija

- Operator `reduce(..)` kombinuje ("redukuje") listu na jednu vrednost (recimo, skalarnu vrednost kao što su broj ili string).
- Kombinacija/redukcija se apstraktno definiše kao uzimanje dve vrednosti i pravljenje jedne vrednosti od njih.
- Kao i kod mapiranja i filtriranja, način kombinovanja je u potpunosti na Vama i, generalno, zavisan od tipova vrednosti u listi.
- U nekim situacijama, redukcija specificira početnu vrednost, `initialValue`, i radi tako što kombinuje tu početnu vrednost sa prvim elementom u listi i zatim se spušta kaskadno kroz sve ostale vrednosti u listi.

Redukcija: vizualizacija



Redukcija: implementacija

```
const reduce = (array,fn,initialValue) => {
let accumlator;
// proverava se da li postoji početna vrednost
        if(initialValue != undefined)
                accumlator = initialValue;
        else
                accumlator = array[0];
        if(initialValue === undefined)
                for(let i=1;i<array.length;i++)
        accumlator = fn(accumlator,array[i])
        else
                for(const value of array)
                accumlator = fn(accumlator,value)
return [accumlator]
}
```

Redukcija: signatura

```
//  reduce :: (b -> a -> b) -> b -> [a] -> b
var reduce = curry(function(f, x, xs) {
  return xs.reduce(f, x);
});
```

Prvi argument je **funkcija** koja prima tip **b** i tip **a** i vraća tip **b**; drugi argument je nešto tipa **b**; treći argument je **niz vrednosti tipa a**; rezultat je tipa **b**

Apstrakcija

- **Apstrakcija** je “proces razmatranja nečega nezavisno od njegovih asocijacija, atributa ili konkretnih dodataka” (Google dictionary)
- Apstrakcija je pojednostavljivanje:
  - “Jednostavnost je oduzimanje očiglednog i dodavanje smislenog.” John Maeda, “The Laws of Simplicity: Design, Technology, Business, Life”
- Proces apstrakcije sastoji se od dve glavne komponente:
  - **Generalizacija** je proces *pronalaženja sličnosti* (očiglednih) u ponovljenim obrascima i sakrivanje sličnosti iza apstrakcije.
  - **Specijalizacija** je proces korišćenja apstrakcije za kojim se obezbeđuje *samo ono što je različito* (ali smisleno) za pojedinačan slučaj.
- Apstrakcija je proces izdvajanja suštine koncepta.
- Kada se uoči suština problema, vidi se da isto rešenje može da se primeni na različite probleme.

Softver i apstrakcija

- John Vogel Guttag: "Suština apstrakcija je očuvanje informacija koje su relevantne u zadatom kontekstu i zaboravljanje informacija koje su u tom kontekstu nerelevantne.“
- Sam softver je apstrakcija jer je softver
  - Nematerijalna stvar, pa je time apstrahovan od naše mogućnosti da ga svojim čulima direktno opazimo;
  - Uvek namenjen rešavanju klase problema, što podrazumeva određivanje klase problema, izdvajanje i očuvavanje relevantnih, odnosno odbacivanje nerelevantnih informacija za tu klasu problema.

Apstrakcija u softverskom inženjerstvu$_{1/3}$

- **Princip apstrakcije:** bazični princip koji ima za cilj smanjenje ponavljanja informacija u programu (obično sa naglaskom na kodu) putem apstrakcija koje obezbeđuju programski jezici ili softverske biblioteke
- Javlja se u dva oblika:
  - kao preporuka programerima,
  - Kao zahtev programskog jezika.
- Dve formulacije principa apstrakcije
  - Šmitova: "The structure of typed programming languages"): "Fraza bilo koje semantički smislene semantičke klase treba da bude imenovana.".
  - Pirsova: "Svaki značajan deo funkcionalnosti u programu trebao bi da bude implementiran samo na jednom mestu u izvornom kodu. Kada se slične funkcije izvršavaju u različitim delovima koda, generalno je delotvorno kombinovati ih (delove koda) u jedno tako što će se promenljivi delovi apstrahovati."

Apstrakcija u softverskom inženjerstvu$_{2/3}$

- **Proces** uklanjanja fizičkih, prostornih ili vremenskih detalja ili atributa u proučavanju objekata ili sistemima da bi se pažnja usmerila na detalje veće važnosti; ima dve glavne komponente:
  - **Generalizacija** je pronalaženje *sličnosti* (očiglednih) koje se ponavljaju u obrascima i sakrivanje sličnost iza apstrakcije.
  - **Specializacija** je izdvajanje *onoga što je različito* (smisleno različito) u svakom od slučajeva koji poseduju međusobne sličnosti.
- **Apstraktni koncepti-objekti** koji odslikavaju zajednička svojstava ili atributa različitih ne-apstraktnih objekata ili sistema - rezultat procesa apstrakcije.

Apstrakcija u softverskom inženjerstvu$_{3/3}$

- Apstrakcija je osnova za implementaciju jednog od osnovnih principa arhitekture softvera - PONOVNE UPOTREBLJIVOSTI (engl. reusability)
- Apstrakcija može da DOPRINESE I POJEDNOSTAVLJENJU PROBLEMA tako što će se problem sagledati kroz proces izdvajanja suštine i koncepata.
- U oblasti softvera ističe se pristup koji se zove modelom-vođen razvoj softvera koji se u velikoj meri oslanja na apstrakciju. To se, naravno, kasnije proširuje i na izradu koda čime se značajno smanjuje potreba za izradom novog koda.

Osnove i mehanizmi apstrakcije u softveru 1/2

- ŽIVOTNI CIKLUS SOFTVERA je proces koji obuhvata sledeće faze: definisanje zahteva, dizajn, kodiranje (implementacija), uvođenje u rad i održavanje
- ŽIVOTNI CIKLUS SOFTVERA JE APSTRAKCIJA NAJVIŠEG NIVOA u oblasti softvera.
  - Svaka faza u životnom ciklusu softvera je apstrakcija softvera u toj, određenoj fazi koja se izgrađuje uzimajući u obzir aspekte softvera relevantne za tu fazu.
  - Realizacija faza životnog ciklusa softvera podrazumeva postojanje odgovarajućih mehanizama za apstrakciju u svakoj fazi životnog ciklusa softvera.
  - Ovih mehanizama ima, naravno, mnogo i oni nisu međusobno isključivi. Prosto, ima nekih mehanizama koji se mogu primenjivati i u više faza.

Osnove i mehanizmi apstrakcije u softveru$_{2/2}$

- Osnove apstrakcije u softveru su:
  - Algoritam,
  - Podaci,
  - Jezik.
- Mehanizmi apstrakcije:
  - Tipovi podataka i strukture podataka;
  - Procedure, funkcije;
  - Moduli;
  - Klase;
  - Interfejsi;
  - Razvojna okruženja, ...;

Funkcija kao mehanizam apstrakcije u softveru

- “Ponekad, elegantna implementacija je samo **funkcija**. Ne klasa. Ne radno okruženje. **Samo funkcija**.” John Carmack (Id Software, Oculus VR)
- **Funkcije** su pogodne za apstrakciju zato što poseduju dve odlike od suštinskog značaja za dobru apstrakciju:
  - **Identitet** – Mogućnost dodele imena i ponovno korišćenje u različitim kontekstima.
  - **Kompozibilnost** – Mogućnost da se od jednostavnijih funkcija komponuju složenije funkcije.

Funkcija kao apstrakcija: Primer 1 sa rečima

- Potrebna nam je funkcija koja će iz zadatog skupa reči eliminisati sve reči od 4 slova. Evo takve funkcije:

```
// Izbacivanje reči od 4 slova
const izbacivac = reci => {
const filtrirano = [];
   for (let i = 0, { length } = reci; i < length; i++)
{
      const rec = reci[i];
      if (rec.length !== 4) filtrirano.push(rec);
    }
        return filtrirano;
};
izbacivac(['sunce', 'tele', 'šuma', 'potok']);
```

Funkcija kao apstrakcija: Primer 2 sa rečima

- Potrebna nam je funkcija koja će iz zadatog skupa reči eliminisati sve reči koje ne počinju slovom **s**. Evo takve funkcije:

```
// Izbacivanje reči koje ne počinju sa 's'
const pocinjeSaS = reci => {
const filtrirano = [];
   for (let i = 0, { length } = reci; i < length; i++) {
      const rec = reci[i];
      if (rec.startsWith('s')) filtrirano.push(rec);
    }
        return filtrirano;
};

pocinjeSaS(['sunce', 'tele', 'šuma', 'sveća','potok', 'reka', 'more' ]);
```

Šta je ovde **isto**, a šta **različito**?

```
// Izbacivanje reči koje se sastoje od 4 slova

const censor = reci => {
const filtrirano = [];
   for (let i = 0, { length } = reci; i < length; i++) {
      const rec = reci[i];
      if (rec.length !== 4)
          filtrirano.push(rec);
    }
       return filtrirano;
};
```

```
// Izbacivanje reči koje ne počinju sa 's‘

const pocinjeSaS = reci => {
const filtrirano = [];
   for (let i = 0, { length } = reci; i < length; i++) {
      const rec = reci[i];
      if (rec.startsWith('s'))
         filtrirano.push(rec);
    }
       return filtrirano;
};
```

Šta je isto, a šta različito: reduktor

```
const reduce = (reducer, initial, arr) => {
// zajednički deo
   let acc = initial;
   for (let i = 0, { length } = arr; i < length; i++) {
   // jedinstven deo u reduktorskom pozivu
      acc = reducer(acc, arr[i]); // reducer() rukuje različitošću
  // još zajedničkog
   }
      return acc;
};

alert (reduce((acc, curr) => acc + curr, 0, [1,2,3])); // 6
alert (reduce((acc, curr) => acc - curr, 0, [1,2,3])); // -6
```

Reduktor: uraditi više sa manje koda 1/2

```
const reduce = (reducer, initial, arr) => {
  let acc = initial;
   for (let i = 0, { length } = arr; i <
        length; i++) {
      acc = reducer(acc, arr[i]);
   }
      return acc;
};
```

Reduktor: uraditi više sa manje koda$_{2/2}$

```
const filter = (fn, arr) =>
   reduce((acc, curr) => fn(curr) ?
   acc.concat([curr]) : acc, [], arr);

const izbacivac = reci =>
   filter( rec => rec.length !== 4,reci);

const pocinjeSaS = reci =>
  filter( rec => rec.startsWith('s'),reci);

console.log (izbacivac(['oops', 'gasp', 'shout', 'sun']));
console.log (pocinjeSaS(['oops', 'gasp', 'shout', 'sun']));
```

Funkcija kao apstrakcija: operisanje nad različitim tipovima podataka

- Na primer, `.filter()` ne mora da operiše samo nad nizovima stringova, može da filtrira i brojeve ako mu se prosledi funkcija koja zna da radi sa tim tipom podatka.
- Primer:

```
const veciJednak = granica => n => n >= granica;

const gte2 = veciJednak(2);
console.log([1, 2, 3, 4].filter(gte2));
const gtea = veciJednak('a');
console.log(['a','A', 'b', 'w'].filter(gtea));
```

Funkcija kao apstrakcija: sakrivanje (nepotrebnih) detalja$_{1/2}$

- Apstrakcija je mehanizam za sakrivanje (nepotrebnih) detalja:

```
const forEach = (array,fn) => {
    let result = [];
    for (let i=0; i<array.length; i++){
        result [i] = fn(array[i])
    }
        return result;
}

let outArray = forEach([1,2,3], (value) => 2*value)
console.log (outArray) //=> ([2,4,6]
outArray = forEach([10,20,30], (value) => 2+value)
console.log (outArray) //=> ([12,22,32]
```

Funkcija kao apstrakcija: sakrivanje (nepotrebnih) detalja$_{2/2}$

```
// funkcija forEachObject poziva funkciju fn nad
// svojstvom i njegovom vrednošću kao argumentima
const forEachObject = (obj,fn) => {
   for (var property in obj) {
       if (obj.hasOwnProperty(property)) {
          fn(property, obj[property])
       }
    }
}

let mojObjekat = {a:1,
                  b:2}
let mojObjekat1 = {prvoSvojstvo:' Ja sam vrednost prvog svojstva',
                   drugoSvojstvo: mojObjekat}

console.log (' Poziv funkcije forEachObject za objekat mojObjekat')
forEachObject(mojObjekat, (k,v) => console.log(k + ":" + v))

console.log (' Poziv funkcije forEachObject za objekat mojObjekat1')
forEachObject(mojObjekat1, (k,v) => console.log(k + ":" + v))
```

Nešto upotrebljivo: funkcija **every**

- Funkcija `every()` koja ima dva argumenta: **niz** i **funkciju** proverava da li se svi elementi niza evaluiraju na `true` putem prosleđene funkcije.

```
const every = (arr,fn) => {
   let result = true;
   for(let i=0; i<arr.length; i++)
     result = result && fn(arr[i])
   return result
}

const every1 = (arr,fn) => {
   let result = true;
   for(const value of arr)
     result = result && fn(value)
   return result
}
const jeParan   = x => x % 2 == 0 ? true : false
const jeNeparan = x => x % 2 != 0 ? true : false
console.log (every([1,2,3], jeParan))
console.log (every([2,4,6], jeParan))
console.log (every([1,2,3], jeNeparan))
console.log (every1([1,7,3], jeNeparan))
```

Još nešto upotrebljivo: funkcija `some`

- Funkcija `some()` radi obrnuto od funkcije `every()`, vraća `true` ako se bar jedan od elemenata niza evaluira na `true` pri primeni prosleđene funkcije.

```
const some = (arr,fn) => {
    let result = false;
    for(const value of arr)
      result = result || fn(value)
    return result
}
```

Nešto veoma potrebno: funkcija `sort`

- Pretpostavimo da treba sortirati sledeću listu :

```
var fruit = ['cherries', 'apples', 'bananas'];
```

- Za to postoji ugrađena funkcija sort sa sledećom signaturom:

```
arr.sort([compare])
```

- Sve što upravlja sortiranjem (kriterijum po kome se sortira, uređenje - opadajuće, rastuće, ...) ugrađeno je u funkciju `compare`.

Kostur funkcije **compare**

```
function compare(a, b) {
  if (a je manje od b po nekom kriterijumu uređivanja) {
    return -1;
  }
  if (a je veće od b po nekom kriterijumu uređivanja) {
    return 1;
  }
  // ukoliko nije ništa od prethodnog, mora da bude a=b

  return 0;
}
```

Pozivanje funkcije za sortiranje

```
var people = [
{firstname: "aaFirstName", lastname: "cclastName"},
{firstname: "ccFirstName", lastname: "aalastName"},
{firstname: "bbFirstName", lastname: "bblastName"}
];
people.sort((a,b) => { return (a.firstname < b.firstname) ? -1 :
(a.firstname > b.firstname) ? 1 : 0 })
```

vratiće:

```
[{ firstname: 'aaFirstName', lastname: 'cclastName' },
 { firstname: 'bbFirstName', lastname: 'bblastName' },
 { firstname: 'ccFirstName', lastname: 'aalastName' }]
```

- Šta tu nije baš najbolje?
  - U našoj funkciji `compare()` hardkodiran je kriterijum sortiranja `firstName`. Ovakvu funkciju ne možemo da iskoristimo čak ni za sortiranje po kriterijumu `lastName`

Da malo popravimo

```
const sortBy = (property) => {
  return (a,b) => {
    var result = (a[property] < b[property]) ? -1 :
                 (a[property] > b[property]) ? 1 : 0;
     return result;
    }
}
```

- Funkcija sortBy prima jedan argument sa imenom `property` i vraća novu funkciju sa dva argumenta:

```
. . .
return (a,b) => {// telo funkcije }
. . .
```

- Telo vraćene funkcije izražava logiku funkcije `compare`:

```
. . .
(a[property] < b[property]) ? -1 : (a[property] > b[property]) ? 1 : 0;
. . .
```

Šta nam je dala mala popravka

- Možemo da sortiramo po jednom ili drugom kriterijumu (ili nekom trećem) ako napišemo odgovarajući poziv funkcije sortBy():
  - **`people.sort(sortBy("firstName"))`**
  - **`people.sort(sortBy("lastName"))`**

SortBy i zatvaranje: funkcija SortBy

```
const sortBy = (property) => {
    return (a,b) => {
        var result = (a[property] < b[property]) ?
        -1 : (a[property] >
        b[property]) ? 1 : 0;
            return result;
    }
}
```

- sortBy vraća kontekst
  - property = "prosledjenaVrednost" - od roditelja
  - (a,b) => { /* implementacija nove funkcije */ }

Kompozicija i razvoj softvera$_{1/2}$

- **Kompozicija**: “Akt kombinovanja delova ili elemenata da bi se formirala celina.”
- Proces razvoja softvera je razlaganje složenih problema na manje probleme, izgradnja komponenti za rešavanje tih manjih problema, i zatim komponovanje tih komponenti da bi se formirala kompletna, složenija aplikacije.
- Paul Chiusano, Rúnar Bjarnason (2014). Functional Programming in Scala, Manning Publications:
  - "Modularnost je svojstvo da se Vaš softver može razdvojiti u svoje sastavne delove i da se ti delovi mogu ponovo koristiti nezavisno od celine, na nove načine koje niste predvideli kada ste te delove kreirali."
  - "Kompozicionalnost je odlika da Vaš softver može da se razume kao celina kroz razumevanje njegovih delova i pravila koja upravljaju kompozicijom.“

Kompozicija i razvoj softvera$_{2/2}$

- Dobra kompozicija je najvažnija odlika dobrog softvera.
- **Kompozicija je suština razvoja softvera:** Za razvoj softvera **neophodne su nam komponente** i **mehanizmi za njihovu adekvatnu kompoziciju**
- Kompozicija softvera nastoji da obezbedi mehanizme za **sistematično konstruisanje** bazirano na **dobro definisanim** jedinicama softvera.

Dobro definisana jedinica: softverska komponenta

- Softverska komponenta je element softvera koji se može **nezavisno koristiti** i **kombinovati** sa drugim elementima.
- Softverska komponenta je softverska jedinica funkcionalnosti koja **upravlja pojedinačnom apstrakcijom**.

Mehanizmi za komponovanje softvera

- Mehanizmi za komponovanje softvera su brojni i različiti.
- Pomaže ako se klasifikuju
  - Jedna klasifikacija je **prema pogledu:**
    - programerski,
    - konstrukcioni, i
    - komponentno-baziran razvoj
  - Druga klasifikacija je **opšta (nezavisna od pogleda):**
    - sadržavanje (eng. *Containment*)
    - proširivanje (engl. *extension*)
    - povezivanje (engl. *connection*
    - koordinacija (engl. *Coordination*)

Mehanizmi prema pogledu: programerski pogled

- Svaki legitimni konstrukt programskog jezika se smatra jedinicom kompozicije, a sama kompozicija je prosto združivanje tih konstrukata u druge konstrukte definisane u programskom jeziku.
  - Smislene jedinice kompozicije u ovom pogledu su, na primer, funkcije u funkcionalnim jezicima, procedure u imperativnim jezicima, klase u objektnim jezicima, aspekti u aspektno-orijentisanim jezicima, itd.

Mehanizmi prema pogledu: konstrukcioni pogled

- Jedinice kompozicije nazivaju se komponentama, pri čemu su komponente labavo definisane jedinice softvera sa priključčima koji predstavljaju tačke interakcije ili povezivanja.
  - Komponente mogu da budu proizvoljne jedinice softvera koje se mogu povezati nekim mehanizmima kao što su moduli povezani jezikom interakcije modula, ili Java Bean-ovi povezani korišćenjem jezika Piccola, itd.

Mehanizmi prema pogledu: komponentni razvoj

- Jedinice kompozicije su precizno definisane putem ***komponentnog modela***.
  - Primenu je našao u servisno-orijentisanim arhitekturama softvera poput web servisa
- Komponentni model definiše ***šta su komponente*** (njihovu sintaksu i semantiku) i ***koji se operatori kompozicije mogu koristiti*** za njihovo komponovanje.
  - Komponenta je "element softvera koji je saglasan sa komponentnim modelom i može se nezavisno razvijati i komponovati bez modifikacije u skladu sa standardima kompozicije".
  - Primeri operatora su koreografija i orkestracija.

Opšti mehanizmi: Sadržavanje

- Mehanizmi sadržavanja vrše postavljanje jedinica ponašanja (softverskih komponenti) unutar definicije veće jedinice
- Primeri:
  - ugnježdene definicije funkcija,
  - procedure,
  - moduli i klase,
  - kompozitni objekti;

Opšti mehanizmi: Proširivanje

- Mehanizmi proširivanja definišu ponašanja jedinice (softverske komponente) njenim proširivanjem putem najmanje dve druge jedinice kompozicije
- Primeri:
  - višestruko nasleđivanje u OO pristupu,
  - preplitanje aspekata u aspektnom programiranju.

Opšti mehanizmi: Povezivanje

- Definisanje ponašanja (softverske komponente) koje je interakcija između ponašanja više jedinica.
- Primeri: direktno i indirektno prosleđivanje poruka.

Opšti mehanizmi: Koordinacija

- Definisanje ponašanja (softverske komponente) koje proističe iz koordinacije ponašanja više jedinica.
- Primeri:
  - prostori torki za podatke,
  - orkestracija servisnih komponenti.

Označivač

Programski jezik

- Programski jezik sastoji se od *gramatike/sintakse* i *modela izvršavanja* (engl. *execution model*).
- Model izvršavanja specificira ponašanje elemenata jezika, odnosno predstavlja opis semantike programskog jezika.
- Model izvršavanja obuhvata stvari kao što su nedeljiva jedinica rada i ograničenja koja opisuju redosled kojim se dešavaju te jedinice izvršavanja.
- Model izvršavanja implementira se putem kompajlera ili interpretera i često uključuje i sistem izvršavanja (engl. *runtime system*).
  - Kompajler implementira unapred definisan redosled (statički), a sistem izvršavanja dinamički redosled.
  - Interpreter predstavlja kombinaciju translatora i sistema izvršavanja i omogućuje kombinovanje statičkog i dinamičkog određivanja redosleda.

Paradigme programiranja

- **Paradigma programiranja** je termin koji se koristi da označi klasifikaciju programskih jezika baziranu na svojstvima jezika.
- Klase koje se najčešće koriste nisu disjunktne, odnosno isti programski jezik može da se klasifikuje u više klasa.
- Uobičajena podela na najvišem nivou programske jezike i danas klasifikuju u dve paradigme:
  - (1) imperativna, i
  - (2) deklarativna.

Imperativno programiranje

- Hardverska implementacija računara koje mi danas koristimo je imperativna. Računarski hardver je dizajniran da izvršava *mašinski kod* koji je nativan za računar i pisan je imperativnim stilom.
  - Stanje programa (sadržaj memorijskih lokacija u datom vremenskim trenutku izvršavanja programa) je definisano sadržajem memorije, a naredbe su instrukcije u nativnom mašinskom jeziku računara - instrukcije koje hardver računara može da izvršava.
- ***Imperativno programiranje*** je stil programiranja koji opisuje računanje putem naredbi koje menjaju stanje programa.
- Imperativno programiranje usretsređuje se na opisivanje ***kako*** program radi.

Deklarativno programiranje

- ***Deklarativno programiranje*** u računarskoj nauci je stil programiranja koji izgrađuje strukturu i elemente računarskog programa tako da oni izražavaju logiku sračunavanja, bez opisivanja toka kontrole sračunavanja.
- Deklarativni stil opisuje ***šta*** program treba da uradi, dok se deo koji opisuje ***kako*** se to radi prepušta implementaciji jezika.
- To je stil programiranja gde programi opisuje svoje željene rezultate bez eksplicitnog navođenja komandi ili koraka koji moraju da se izvrše.
- Funkcionalno programiranje je deklarativna paradigma

Funkcionalno programiranje: Ideja i koncepti

Šta je funkcionalno programiranje

- ***Funkcionalno programiranje*** **(FP)** je stil programiranja koji **komponuje aplikacije** korišćenjem **čistih funkcija** izbegavajući **mutabilna stanja** i **bočne efekte**.
- U funkcionalnom programiranju **komponente su funkcije** a **mehanizam kompozicije** je **funkcija višeg reda.**

Funkcija

- **Funkcija u matematici**: Funkcija je relacija između skupa (domen) ulaza (argument) i skupa (kodomen) dopustivih izlaza (vrednost funkcije) sa svojstvom da je **svaki ulaz u relaciji sa tačno jednim izlazom**.
  - Funkcija uvek prima ulaz.
  - Funkcija uvek vraća izlaz.
  - Funkcija proizvodi rezultat samo na osnovu ulaza koje prima.
  - Za zadati ulaz postoji samo jedan izlaz funkcije.
- **Funkcija u programiranju generalno**: blok organizovanog, višekratno upotrebljivog koda koji se koristi za izvršavanje pojedinačne akcije. Može, a ne mora da ima ulaz i/ili izlaz (na primer, procedura).
- U **funkcionalnom programiranju**, fokus je korišćenju **koncepta funkcije kakav se javlja u matematici**.

Funkcija: Građanin prvog reda

- U funkcionalnim programskim jezicima funkcije su "građani prvog reda" (engl. "first class citizens").
- To znači da se sa njima, pored pozivanja, može raditi sve ono što se radi sa bilo kojim jezičkim entitetom prve klase (kao što je, na primer, numerik): mogu se dodeljivati varijablama, mogu se prosleđivati kao argumenti drugim funkcijama, mogu da budu povratne vrednosti drugih funkcija.
- Funkcija može da primi i vrati vrednost bilo kog tipa.
- Funkcija koja prima i/ili vraća jednu ili više drugih funkcija ima posebnu ulogu u funkcionalnom programiranju i posebno ime - *funkcija višeg reda*.

Funkcija: komuniciranje ulaznih podataka

- Dva termina: *argument* i *parametar*. Ponekad, ova dva termina se (neopravdano) tretiraju kao sinonimi.
  - Termin *argument* označava vrednosti koje se prosleđuju funkciji,
  - Termin *parametar* označava imenovani entitet/varijablu unutar funkcije koja tu prosleđenu vrednost prihvata.
  - Broj argumenata koje funkcija očekuje određen je brojem deklarisanih parametara. Ovaj broj se naziva ***arnost*** funkcije i moguće ga je pribaviti u toku izvršavanja programa. Ta dva broja (broj parametara i broj argumenata) u opštem slučaju ne moraju da budu jednaka.

```
function imeFunkcije(x,y) { // x i y su parametri
  // Ovde dolazi telo funkcije - kod koji se
  // izvršava pri pozivu funkcije
}
var a = 3;
imeFunkcije(a, a *2);// a, a*2 su argumenti
```

Funkcija: komuniciranje izlaznih podataka

- Funkcija **bi uvek trebala** da **ima povratnu vrednost**.
- Funkcija **može da vrati maksimalno jednu vrednost**.
- Više (jednostavnih) vrednosti može se vratiti njihovim organizovanjem u strukturu, poput niza ili objekta.

Funkcija: komuniciranje ulaznih podataka

- Primer : Funkcija 1

```
let procenatVrednost = 5;
let sracunajPorez = (value) => { return
  value/100 * (100 + procenatVrednost) }
alert (sracunajPorez(1500))
```

- Primer : Funkcija 2

```
let procenatVrednost = 5;
let sracunajPorez = (value,percent) => { return
  value/100 * (100 + percent) }
alert (sracunajPorez(1500, procenatVrednost ))
```

- U čemu je razlika između ova dva primera?

Referencijalna transparentnost

- Lingvistika: Kaže se da je kontekst u rečenici referencijalno transparentan ako zamena nekog termina drugim terminom kojim se označava isti entitet ne menja značenje rečenice.
- Funkcija koja za zadati ulaz vraća uvek isti izlaz zadovoljava uslov ***referencijalne transparentnosti*** (RT).
- Referencijalna transparentnost omogućuje model supstitucije u kome se funkcija na svakom mestu može zamenuti svojom vrednošću, na primer:

```
var identity = (i) => { return i } // RT funkcija
sum(4,5) + identity(1) <--> sum(4,5) + 1
```

Bočni efekti

- Bočni efekat je **bilo koje stanje programa koje se može opaziti izvan pozvane funkcije i različito je od povratne vrednosti te funkcije**, n.pr. :
  - Modifikovanje eksterne varijable ili svojstva objekta
  - Logovanje na konzolu
  - Izdavanje izlaza na ekran, u fajl, na mrežu
  - Trigerovanje bilo kog eksternog procesa
  - Poziv druge funkcije koja proizvodi bočni efekat

Stanje programa

- Prosto rečeno, stanje programa je svukupnost sledećih stvari:
  - Tekuće vrednosti svih varijabli,
  - Svi alocirani objekti,
  - Otvoreni deskriptori fajlova,
  - Otvoreni mrežni soketi, itd.
- U stvari, stanje programa su sve informacije koje predstavljaju ono što se trenutno dešava u programu.

Deljeno stanje

- Deljeno stanje je bilo koja varijabla, objekat ili memorijski prostor koji postoji u deljenom dosegu ili kao svojstvo objekta koji se prosleđuje između dosega.
- Deljeno stanje može da uključi globalni doseg ili dosege zatvaranja.
- Problem sa deljenim stanjem je što je za razumevanje efekata funkcije potrebno poznavati kompletnu istoriju svake deljene varijable koju funkcija koristi ili je modifikuje.

Deljeno stanje: primer $1_{1/2}$

```
var globalna = "Originalna vrednost"
alert (globalna)// Ispis: Originalna vrednost
var nevaljalaFunkcija = (value) => {
    globalna = "Izmenjena vrednost";
    return value * 2 }
let Dva = nevaljalaFunkcija (1)
alert (globalna)// Ispis: Izmenjena vrednost
alert(Dva); // ispis: 2
```

Deljeno stanje: primer $1_{2/2}$

```
var globalna = "Originalna vrednost"
alert (globalna)// Ispis: Originalna vrednost
var nevaljalaFunkcija = (value) => {
    let globalna = "Izmenjena vrednost";
    return value * 2 }
let Dva = nevaljalaFunkcija (1)
alert (globalna)// Ispis: Originalna vrednost
alert(Dva); // ispis: 2
```

Deljeno stanje: primer 2

```
// Deljeno stanje
const x = {
val: 2
};
// Mutira (menja) deljeno
  stanje
const x1 = () => x.val += 1;
// Mutira (menja) deljeno
  stanje
const x2 = () => x.val *= 2;
x1();
x2();
console.log(x.val);// Ispis: 6
```

```
/* Ovaj primer je potpuno isti
  kao levi, izuzev  što se
  funkcije pozivaju obrnutim
  redosledom...*/
const y = {
val: 2
};
// Mutira (menja) deljeno
  stanje
const y1 = () => y.val += 1;
// Mutira (menja) deljeno
  stanje
const y2 = () => y.val *= 2;
y2();
y1();
console.log(y.val); // Ispis: 5
```

Problemi sa deljenim stanjem

- Teže je rezonovati o kodu – mora se voditi računa o tome šta drugi mogu da urade nad deljenim stanjem
- Otežano je testiranje – mora se predkonfigurisati deljeno stanje
- Degradiraju se performanse – paralelni procesi konkurišu za isti resurs: deljeno stanje
- Kako ih izbeći?
  - Tako što će se za komuniciranje, kad god je moguće, koristiti parametri a ne eksplicitno deljeno stanje

Deljeno stanje: Primer

```
currentUser = {
name: "John Smith",
balance: 1000,}

const getUserBalance = () => {
   return currentUser.balance }

console.log(getUserBalance()) // 1000
currentUser.balance = 500
console.log(getUserBalance()) // 500
```

Deljeno stanje: Primer izbegavanja deljenog stanja

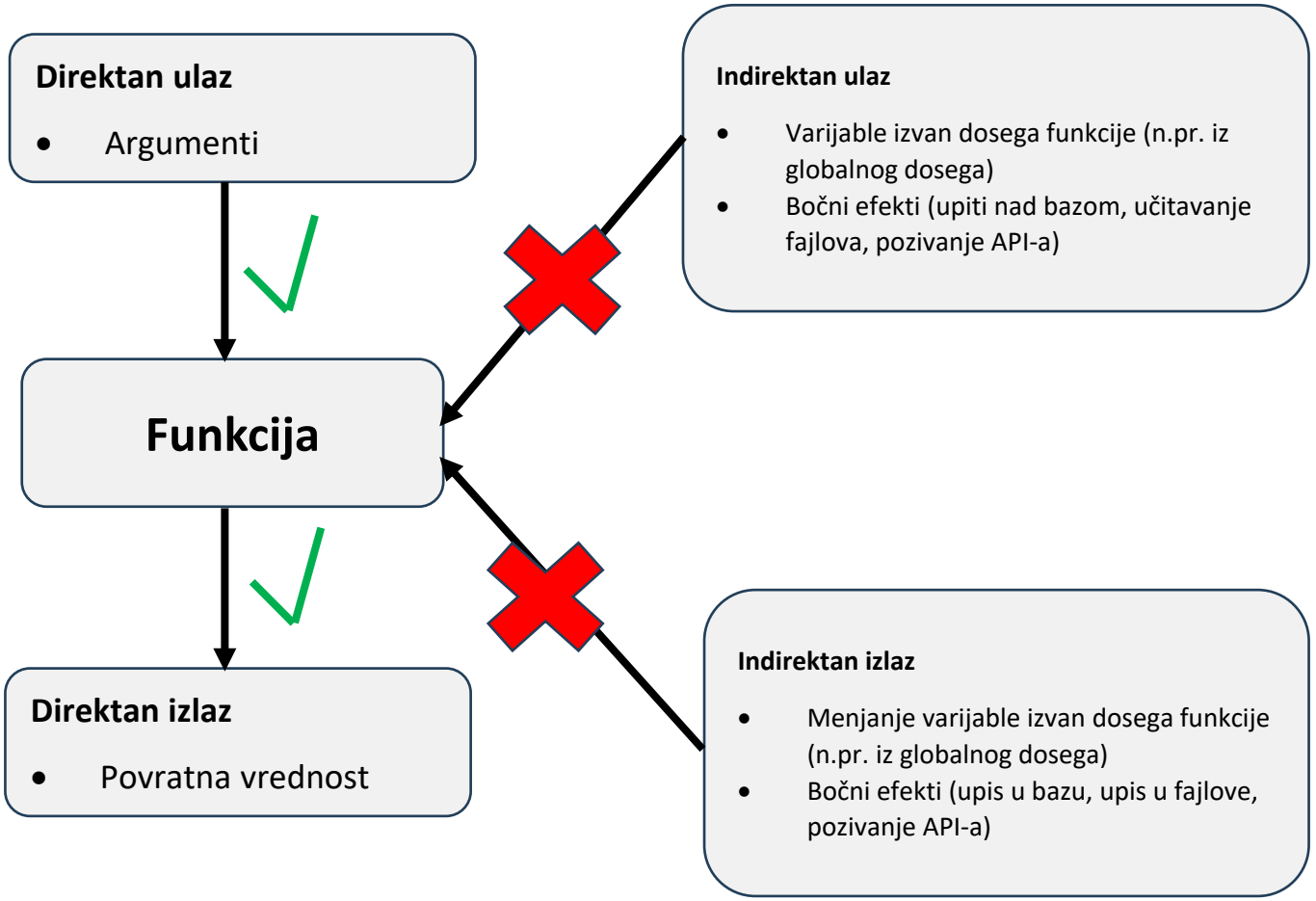
- Tako što će se raditi sa parametrima, ne sa stanjem

```
currentUserObj1 = {
name: "John Smith",
balance: 1000,}

currentUserObj2 = {
name: "Katherine Smith",
balance: 5000,}

function getCurrentUser(userObj) {
  return userObj }

const getUserBalance = user => {
   return user.balance}

let currentUser = getCurrentUser(currentUserObj1)
console.log(getUserBalance(currentUser))
currentUser = getCurrentUser(currentUserObj2)
console.log(getUserBalance(currentUser))
```

Čista funkcija 1/4

- Čista funkcija je funkcija koja:
  - **Zadovoljava uslov referencijalne transparentnosti** (za iste ulaze uvek vraća isti izlaz), i
  - **Nema bočnih efekata**.
- Čista funkcija
  - Prima ulaz samo preko svojih argumenata
  - Za sračunavanje izlaza koristi samo primljeni ulaz
  - Ne menja ništa izvan sebe same

Čista funkcija$_{2/4}$



Čista funkcija 3/4

- Sledeća funkcija je čista funkcija

```
var double = (value) => value * 2;
```

  - Prima ulaz `value` preko liste argumenata
  - Za sračunavanje izlaza koristi samo primljeni ulaz
  - Ne menja ništa izvan sebe same
- Takva funkcija će za isti ulaz (n.pr. 12) UVEK vratiti isti izlaz (24)
- A sledeća funkcija nije čista funkcija

```
var notPure = (value) => { return value/100 * (100 +
  percentValue) }
```

  - Za sračunavanje izlaza koristi primljeni ulaz `value`, ali i vrednost `percentValue` koja je spoljašnja vrednost
- Takva funkcija će za isti ulaz (n.pr. `value` = 12) vratiti izlaz koji će zavisiti od vrednosti `percentValue`

Čista funkcija 4/4

- **Čista funkcija**, zbog mnogih svojih prednosti, **predstavlja temelj funkcionalnog programiranja**.
- Ona je potpuno nezavisna od spoljašnjeg stanja i zbog toga je imuna na česte greške u softveru koje su posledica deljenih stanja.
- Nezavisna priroda čini je izuzetno pogodnom za sledeće stvari:
  - računanja na višeprocesorskim arhitekturama,
  - premeštanje u okviru koda,
  - refaktorizovanje i reorganizovanje
- Njena primena u programima čini te programe fleksibilnijim i manje osetljivim na buduće promene.

Čiste i nečiste funkcije: Redosled pozivanja može da bude važan

- Nečiste funkcije *:*

```
let global = "nešto"
let function1 = (input)
  => {
  /* radi nad varijablom
  input, menja varijablu
  global */
   global = "neštoDrugo"
}

let function2 = () => {
   if(global === "nešto")
   {
     console.log (global)
   }
}
```

- Iste funkcije – “očišćene”*:*

```
let global = "nešto"
let function1 = (input,
  global) => {
  /* radi nad varijablom
  input, menja varijablu
  global */
   global = "neštoDrugo"
}

let function2 = (global) => {
   if(global === "nešto")
   {
     console.log (global)
   }
}
```

Čista funkcija i imutabilnost

- Čiste funkcije NE SMEJU da menjaju eksterna stanja
- Putem parametara funkciji se prosleđuju ulazni podaci. Ti podaci su vidljivi i drugima (minimalno onome ko fukciju poziva).
- Sa tim podacima (argumentima), funkcija može da radi sve što hoće – može i da ih promeni
- Čista funkcija ne sme da menja svoje argumente
  - Napomena: Ako funkcija u JS-u unutar sebe promeni vrednost svog argumenta to ne znači da će se nužno promenuti i stanje kome se može pristupiti izvan funkcije zato što se pri pozivanju funkcije prave lokalne kopije argumenata. Međutim, situacija se komplikuje kada se radi sa objektima.

Čista funkcija ne sme da menja eksterne vrednosti$_{1/2}$

```
let global = {
   vrednost: "nešto",
   }
zapamcenaVrednost = global.vrednost


/* menja vrednost svog argumenta global, znači global iz svog dosega */
let function1 = (global) => {
   global.vrednost = "neštoDrugo iz f1"
}
/* menja vrednost varijable global iz globalnog dosega */
let function2 = () => {
   global.vrednost = "neštoDrugo iz f2"
}
```

Čista funkcija ne sme da menja eksterne vrednosti$_{2/2}$

```
console.log('Očekivao sam ovo: ', global.vrednost, ', a dobio sam ovo: ', global.vrednost );
/* Očekivao sam ovo: nešto, a dobio sam ovo: nešto */
function1(global);
console.log('Očekivao sam ovo: ', global.vrednost, ', a dobio sam ovo: ', global.vrednost );
/* Očekivao sam ovo: neštoDrugo iz f1, a dobio sam ovo: neštoDrugo iz f1 */
function2()
console.log('Očekivao sam ovo: ', zapamcenaVrednost, ', a dobio sam ovo: ', global.vrednost );
/* Očekivao sam ovo: nešto, a dobio sam ovo: neštoDrugo iz f2 */
```

Čista funkcija ne sme da menja svoje argumente$_{1/2}$

```
const getUserBalance = user => {
  return user.balance
}

const rewardUser = user => {
  user.balance = user.balance * 2 // menja argument
  return user // vraća izmenjen argument
}

const getCurrentUser = user => {
   return user
}

const currentUserObj = {
   name: "Current User",
   balance: 2500
   }
const currentUser = getCurrentUser(currentUserObj)
console.log(getUserBalance(currentUser))      // 2500

const rewardedUser = rewardUser(currentUser)
console.log(getUserBalance(rewardedUser))     // 5000
console.log(getUserBalance(currentUser))      // 5000
console.log(getUserBalance(currentUserObj))   // 5000
```

Čista funkcija ne sme da menja svoje argumente$_{2/2}$

```
const getUserBalance = user => {
  return user.balance
}
 const rewardUser = user => {
  return {
    ...user,
    balance: user.balance * 2
  }
}
const getCurrentUser = user => {
   return user
}

const currentUserObj = {
   name: "Current User",
   balance: 2500
   }
const currentUser = getCurrentUser(currentUserObj)
console.log(getUserBalance(currentUser)) //2500
const rewardedUser = rewardUser(currentUser)
console.log(getUserBalance(rewardedUser)) // 5000
console.log(getUserBalance(currentUser)) // 2500
console.log(getUserBalance(currentUserObj)) // 2500
```

Objekti i imutabilnost$_{1/3}$

- Tip `Object` je jedini tip u JavaScript-u koji nije nepromenljiv (imutabilan).
- Iako je, generalno gledano, neprirodno da objekat bude nepromenljiv, ponekad je poželjno da svojstva i objekti budu nepromenljivi.
- Specifikacija ES5 (pa i JavaScript) podržava ovakvo rukovanje objektima na više različitih nijansiranih načina.
- Pri tome, zajedničko za sve pristupe je da kreiraju **plitku nepromenljivost**.
  - To znači da utiču samo na objekat i njegova direktna svojstva. Ako objekat ima referencu na drugi objekat (niz, objekat, funkciju, itd.) sadržaji tog drugog objekta nisu izloženi uticaju ovih mehanizama i ostaju promenljivi.

Objekti i imutabilnost$_{2/3}$

```
const mojNepromenljivObjekat = {}
Object.defineProperty( mojNepromenljivObjekat, "a", {
      value: [1,2,3],
      writable:false,
      configurable:false,
      enumerable:true
} );

mojNepromenljivObjekat.a; // [1,2,3]
mojNepromenljivObjekat.a.push( 4 );
mojNepromenljivObjekat.a; // [1,2,3,4]
```

Objekti i imutabilnost 3/3

```
let mojNiz = [1,2,3]

let mojNepromenljivObjekat = {}
Object.defineProperty( mojNepromenljivObjekat, "a", {
        value: Object.freeze (mojNiz), /* Object.freeze ()
                                          "zamrzava" mojNiz */
        writable:false,
        configurable:false,
        enumerable:true
} );
mojNepromenljivObjekat.a; // [1,2,3]
mojNepromenljivObjekat.a.push( 4 ); /* Greška: Uncaught
TypeError: Cannot add property 3, object is not extensible
*/
```

Objekti i imutabilnost: konstantno objektno svojstvo 1/4

- Kombinovanjem `writable:false` i `configurable:false` može se kreirati **konstantno objektno svojstvo**
  - vrednost mu se ne može izmeniti,
  - samo svojstvo se ne može redefinisati niti obrisati,
  - objektu se mogu dodavati nova svojstva.

Objekti i imutabilnost: konstantno objektno svojstvo 2/4

```
let mojObjekat = {};


mojObjekat.ime = "Ja sam mojObjekat"
mojObjekat. OMILJENI_BROJ = 120;
console.log(JSON.stringify(mojObjekat)) /* {ime: 'Ja sam mojObjekat’,
                                            OMILJENI_BROJ:120} */
Object.defineProperty(mojObjekat,"OMILJENI_BROJ", {
                value:42,
                writable:false,
                configurable:false
        } );
console.log(" Promenuo sam omiljeni broj na: ", mojObjekat.OMILJENI_BROJ, ' i
deskriptore writable i configurable na false.');
console.log(JSON.stringify(mojObjekat)); // {ime: 'Ja sam mojObjekat',
OMILJENI_BROJ:42}
```

Objekti i imutabilnost: konstantno objektno svojstvo 3/4

```
let NOVI_OMILJENI_BROJ = 24
console.log(" Predomislio sam se. Hoću da moj novi omiljeni bude: ", NOVI_OMILJENI_BROJ);


mojObjekat.OMILJENI_BROJ = NOVI_OMILJENI_BROJ;


console.log(" Ali to ne može, jer su mi deskriptori writable i configurable false: ", JSON.stringify(mojObjekat));


delete mojObjekat.OMILJENI_BROJ
console.log('Pokušao sam da izbrišem svojstvo OMILJENI_BROJ, ali to ne može jer su mi deskriptori writable i configurable false: ', JSON.stringify(mojObjekat ))
```

Objekti i imutabilnost: konstantno objektno svojstvo 4/4

```
mojObjekat.MRSKI_BROJ = 142
console.log('Uspešno sam dodao novo svojstvo
MRSKI_BROJ sa vrednošću: ',mojObjekat.MRSKI_BROJ )
console.log('pa sada izgledam ovako:',
JSON.stringify(mojObjekat))
console.log('sa deskriptorima):')
descriptor = Object.getOwnPropertyDescriptor(
mojObjekat, "OMILJENI_BROJ" );
console.log(JSON.stringify(descriptor, null, 2 ));
descriptor = Object.getOwnPropertyDescriptor(
mojObjekat, "MRSKI_BROJ" );
console.log(JSON.stringify(descriptor, null, 2 ));
```

Objekti i imutabilnost: sprečavanje proširivanja objekta

- Sprečavanje dodavanje novih svojstava objektu, a da to ne utiče na ostala svojstva objekta postiže se metodom `Object.preventExtensions()`:

```
var mojObjekat = {
   a:2
};

Object.preventExtensions( mojObjekat );

mojObjekat.b = 3;
mojObjekat.b; // undefined
```

Objekti i imutabilnost: pečaćenje objekta

- Funkcija `Object.seal()` kreira "zapečaćen" objekat
  - Uzima se postojeći objekat i, u suštini, poziva `Object.preventExtensions()` nad tim objektom čime se sva njegova postojeća svojstva označavaju kao `configurable:false`.
- Nakon operacije pečaćenja, ne samo da ne mogu da se dodaju nova svojstva, već ne može ni da se rekonfiguriše ili briše ništa od postojećih svojstava
- Jedino još uvek mogu da se modifikuju vrednosti svojstva ako im deskriptor `writable` nije imao vrednost `false`.

Pečaćenje objekta primer 1/3

```
var mojObjekat = {
        a:2
};
console.log(JSON.stringify(mojObjekat)) // {a:2}

let descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "a" );
console.log(JSON.stringify(descriptor, null, 2))
/*
{
  "value": 2,
  "writable": true,
  "enumerable": true,
  "configurable": true
} */
```

Pečaćenje objekta primer 2/3

```
var mojObjekat = {
        a:2
};
console.log(JSON.stringify(mojObjekat)) // {a:2}

let descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "a" );
console.log(JSON.stringify(descriptor, null, 2))
/*
{
  "value": 2,
  "writable": true,
  "enumerable": true,
  "configurable": true
} */
```

Pečaćenje objekta primer 3/3

```
Object.seal( mojObjekat );
console.log(JSON.stringify(mojObjekat)) // {a:2}
descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "a" );
console.log(JSON.stringify(descriptor, null, 2))
/*
{
  "value": 2,
  "writable": true,
  "enumerable": true,
  "configurable": false
} */

mojObjekat.b = 3;
console.log(JSON.stringify(mojObjekat)) // {a:2}
descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "b" ); // undefined
console.log(JSON.stringify(descriptor, null, 2)) // undefined
```

Objekti i imutabilnost: zamrzavanje objekta

- Metoda `Object.freeze()` kreira "zamrznuti" objekat na sledeći način:
  - Uzima postojeći objekat ii nad tim objektom poziva `Object.seal()`.
  - Dodatno markira sva svojstva "pristupa podacima" kao `writable:false`, tako da se vrednosti svojstava ne mogu menjati.
- Zamrzavanje je najviši nivo nepromenljivosti koji se može primeniti na sam objekat jer se njime sprečava svaka promena objekta i svaka promena njegovih direktnih svojstava (sadržaji bilo kojih drugih referenciranih objekata nisu izloženi ovom uticaju).
  - Moguće je i "duboko zamrznuti" objekat pozivajući `Object.freeze()` nad tim objektom i, zatim, rekurzivno, iterirajući nad svim objektima koje on referencira pozivati `Object.freeze()`.
  - Pri tome treba biti krajnje oprezan jer se može nenamerno uticati na druge (deljene) objekte i time izazvati vrlo neprijatni ukupni efekti.

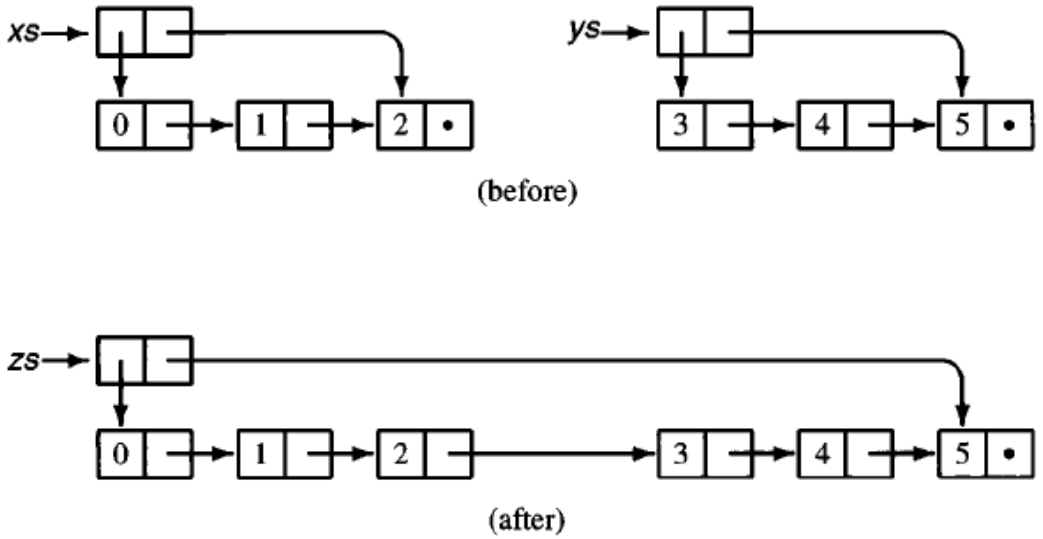
Funkcije su objekti

```
const immutFun = x => x
let nonImmutFun = immutFun
console.log (immutFun(2)) // 2
console.log (nonImmutFun(2)) // 2
nonImmutFun = x => 3*x
console.log (nonImmutFun(2)) // 6
immutFun = x => 3*x // Greška
```

Čiste funkcionalne strukture podataka

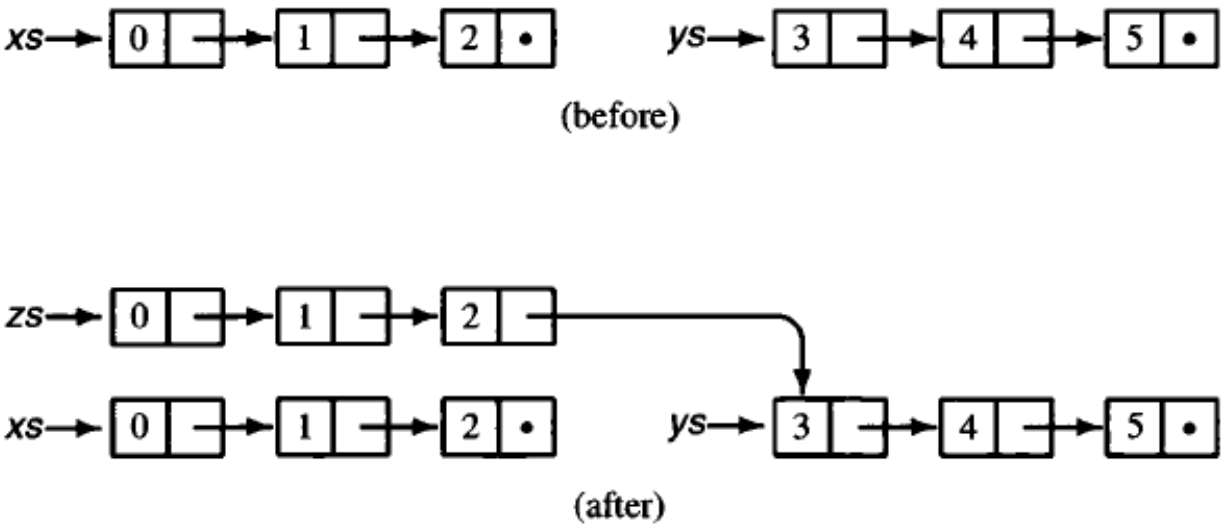
- Da bi se zadovoljio princip imutabilnosti, čiste funkcionalne strukture moraju da imaju svojstvo *prezistencije*, svojstvo koje sprečava da se modifikuju prethodne verzije strukture podataka.
- Iz tog razloga se čiste funkcionalne strukture podataka često predstavljaju na drugačiji način od odgovarajućih imperativnih struktura.
  - Na primer, niz sa konstantnim vremenom pristupa i ažuriranja je osnovna komponenta u većini imperativnih jezika i mnoge imperativne strukture podataka poput heš-tabela i binarnog hipa su bazirane na nizovima. Takva reprezentacija nema svojstvo perzistencije.
  - Nizovi se mogu zameniti mapama ili listama sa slučajnim pristupom što dozvoljava čistu funkcionalnu implementaciju, ali je cena logaritamsko a ne konstantno vreme pristupa i modifikacije.

Spajanje lista u imperativnoj postavci



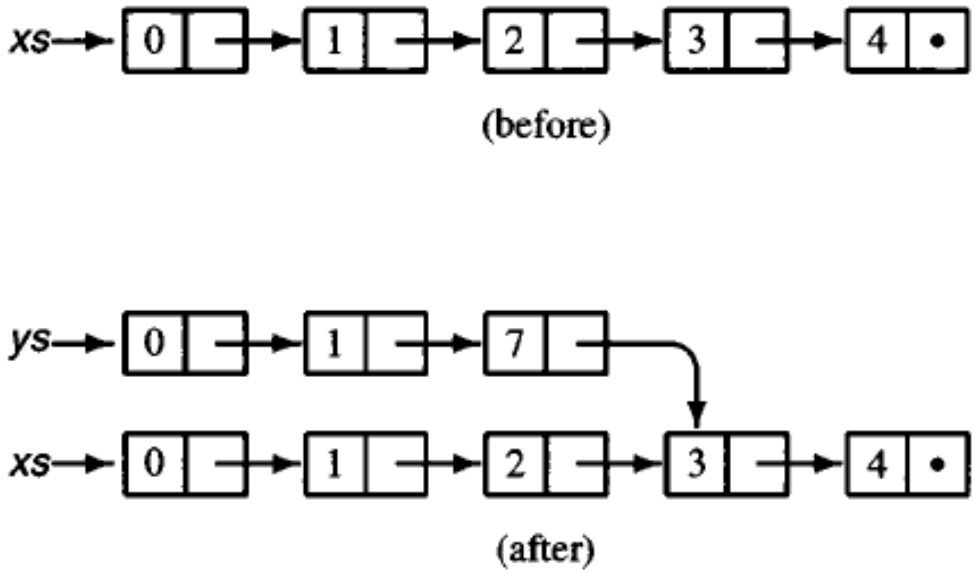
Ovde se prosto modifikuje pokazivač iz poslednjeg elementa prve liste tako da pokazuje na prvi element druge liste. Zauzeće memorije je isto i nema kopiranja – samo se ažuriraju pokazivači. Cena je što ta operacija uništava oba svoja argumenta (xs i ys), nakon spajanja gube se informacije o listama xs i ys i one se ne mogu ponovo koristiti.

Spajanje lista u funkcionalnoj postavci



Ovde liste zs i ys dele čvorove list ys, dakle lista ys se ne kopira, dok se lista xs kopira u novu listu zs. Ovde se zadržavaju sve informacije o argumentima xs i ys tako da se oni mogu ponovo koristiti. Cena je kopiranje liste xs.

Ažuriranje elementa liste u funkcionalnoj postavci



Na slici je prikazana operacija ys = update(xs, 2, 7). Ovde postoji deljenje čvorova između lista xs i ys. I ovde se zadržavaju sve informacije o argumentu xs i tako da se on može ponovo koristiti – zna se da je ulazna lista imala čvorove 0,1,2,3,4 dok izlazna lista ima čvorove 0, 1, 7, 3, 4. Kopirani su čvorovi liste xs koji prethhode modifikovanom čvoru i sam modifikovani čvor.

Čiste funkcionalne strukture podataka: memoizacija

- Memoizacija je jedan od ključnih instrumenata za pravljenje čistih funkcionalnih struktura podataka (jednom sračunato se ne računa ponovo, već se preuzima iz skladišta)
- Primer: "Skupa" funkcija `skupaFun()` koju treba pozivati više puta sa istim ulazom
- Memoizacija:

```
let mojeSkladiste = { 2 : 3, 4 : 5 }
const skupaFun = x => x+1
/* Proveri da ima ključeva u sFunSkladiste; ako ima, preuzmi postojeći rezultat
   iz sFunSkladiste, inače ažuriraj objekat sFunSkladiste novim ključem i
   vrednošću */
const memoise = (sFunSkladiste,ip) =>
sFunSkladiste.hasOwnProperty(ip) ?
sFunSkladiste[ip] : sFunSkladiste[ip] = skupaFun(ip)

console.log (memoise (mojeSkladiste, 2)) // 3
ispisiObjekat(mojeSkladiste) // { 2 : 3, 4 : 5 }
console.log (memoise (mojeSkladiste, 7)) // 8
ispisiObjekat(mojeSkladiste)// { 2 : 3, 4 : 5, 7:8 }
```

Imutabilnost: nepromenljivost vrednosti

- Nepromenljivost vrednosti ne znači apsolutno nepromenljivu vrednost. U pitanju je princip stvaranja i praćenja novih vrednosti kako se stanje programa menja, umesto da se nepovratno menjaju postojeće vrednosti.
- Ovaj pristup dovodi do veće pouzdanosti u čitanju koda, jer se ograničavaju mogućnosti da se stanje programa promeni na načine koje ne vidimo ili ne očekujemo.
- Za čitljivost koda manje je važna nemogućnosti da se promeni vrednost, mnogo je važnija disciplina da se vrednost tretira kao nepromenljiva.

Imutabilnost: kako je postići

- Generalno, u kodnoj bazi dogmatski obezbediti da funkcije vraćaju nove objekte, umesto da mutiraju svoje argumente
  - Jedini stvarni razlog za mutiranje je performansa.
- Obično se precenjuje sposobnost `const` deklaracije (konstante) u JS-u da signalizira namere i nametne nepromenljivost (na primer const deklaracija objekta ne znači ništa).
- `Object.freeze()` je dobar ugrađeni način postavljanja plitke nepromenljivosti na nizu ili objektu.
- Za programe koji zahtevaju visoke performanse ili u slučajevima čestih promena, kreiranje novog niza ili objekta (naročito ako sadrži puno podataka) je nepoželjno, kako zbog obrade tako i zbog memorije.
  - Alternativa je i korišćenje eksternih alata koji obezbeđuju imutabilne kolekcije tako da štite od mutiranja podataka uz korišćenje struktura podataka na način koji poboljšava performansu
  - Jedan primer je Facebook-ova biblioteka Immutable.js

Komponovanje

- "U računarskoj nauci, **kompozitni tip je bilo koji tip** koji se može konstruisati u programu korišćenjem **primitivnih tipova** programskog jezika i/ili **drugih kompozitnih tipova**. […] Akt konstruisanja kompozitnog tipa poznat je kao komponovanje." *Wikipedia.*
- Drugi kompozitni tipovi čije komponovanje ćemo demonstrirati su:
  - Funkcije
  - Objekti

Komponovanje: komponovanje funkcija

- Komponovanje funkcija je proces primene funkcije na drugu funkciju.
- Matematička notacija je:

Za dve zadate funkcije *f i g, kompozicija je*

$$(f \circ g)(x) = f(g(x))$$

simbol ◦ označava kompoziciju a značenje je: prvo se evaluira (sračunava) $g(x)$ pa se na rezultat primenjuje funkcija f (dobijeni izlaz se prosleđuje kao argument funkciji f)

Komponovanje funkcija: nejjednostavniji primer

```
const g = n => n + 1;
const f = n => n * 2;
const uradiPosao = x => {
   const nakonG = g(x);
   const nakonF = f(nakonG);
      return nakonF;
   };
alert (' Rezultat je: ' + uradiPosao(20));
 // Ispis: Rezultat je: 42
```

Komponovanje funkcija: šta zahteva

- Komponovanje funkcija zahteva da funkcija može da primi drugu funkciju kao svoj argument
- Način na koji JS tretira funkcije zadovoljava ovaj zahtev
- Zbog toga je JS jezik koji je POGODAN za funkcionalno programiranje

Komponovanje funkcija: primer

- Dat je niz arr = [1, 2, 3, 4], želimo da iz njega **izvadimo sve neparne brojeve** i da **sračunamo zbir** brojeva preostalih u nizu.
- Šta bismo radili:
  - **Prvo** bismo **isfiltrirali** (izvadili) sve neparne
  - **Zatim** bismo **sabrali** preostale
- Trebaju nam dve funkcije:
  - **Filtriranje** (da povadimo neparne) - `filterOutOdd`
  - **Sumiranje** (da saberemo preostale) - `add`
- Konačan rezultat dobićemo tako što ćemo na naš niz prvo da primenimo funkciju filtriranja `filterOutOdd` pa na dobijeni rezultat funkciju sumiranja `add`
- U stvari, na ulazni niz ćemo da primenimo kompoziciju funkcija `filterOutOdd` i `add`

Komponovanje funkcija: primer

```
const jeParan = x => x % 2 === 0
const izbaciNeparne = kolekcija =>
  kolekcija.filter(jeParan)

const dodaj = (x, y) => x + y
const suma = kolekcija =>
  kolekcija.reduce(dodaj)

const sumaParni = kolekcija =>
  suma(izbaciNeparne(kolekcija))

sumaParni([1, 2, 3, 4])// 6
```

Komponovanje funkcija: malo opštije

- Šta se, u stvari, dešava:

```
const kompozicijaFunkcija = ulazniObjekat =>
  drugaFunkcija(prvaFunkcija(ulazniObjekat))
```

- Dakle, funkcija koja implementira kompoziciju (dve) funkcije je:

```
const komponuj = (fn1, fn2) => ulazniObjekat
  => fn1(fn2(ulazniObjekat))
```

Komponovanje funkcija: kako sada izgleda primer

```
const jeParan = x => x % 2 === 0
const izbaciNeparne = kolekcija =>
  kolekcija.filter(jeParan)

const dodaj = (x, y) => x + y
const suma = kolekcija =>
  kolekcija.reduce(dodaj)

const sumaParni = kolekcija =>
  komponuj(suma, izbaciNeparne)(kolekcija)
sumaParni([1, 2, 3, 4])// 6
```

Komponovanje: komponovanje objekata

- Komponovanje objekata je akt izgradnje novog objekta na bazi postojećih primitivnih i kompozitnih tipova u jeziku.
- Primer
  - Komponente (primitive tipa `String`):

```
const ime = 'Claude';
const prezime = 'Debussy';
```

  - Kompozit (tip `Object`):

```
const punoIme = {
ime:'Claude',
prezime:'Debussy'
};
```

Komponovanje objekata: tri fundamentalne tehnike

- **Konkatenacija** kompozitni objekat formira **dodavanjem novih svojstava postojećem objektu**. Na ovaj način formira se **novi objekat** gde **podobjekti ne zadržavaju svoj identitet** pa se ne mogu destrukturirati iz agregacije.
- **Agregacija** kompozitni objekat formira iz prebrojive kolekcije podobjekata tako da **kompozitni objekat *sadrži* podobjekte**. Pri tome, **svaki podobjekat zadržava svoj identitet** pa se može destrukturirati iz agregacije bez gubitka informacija.
- **Delegiranje** kompozitni objekat formira tako što dati objekat delegira/ prosleđuje drugom objektu.

- Ove tri tehnike **nisu međusobno isključive –** delegiranje je podskup agregacije, konkatenacija se može koristiti za formiranje delegata i agregata, itd.

Komponovanje konkatenacijom

- Jedan način za konkatenaciju objekata je korišćenje funkcije `Object.assign()`:

```
let ObjekatD = Object.assign({}, ObjekatA, ObjekatB);
```

- Drugi način je `spread` operator:

```
let ObjekatD  =  {...ObjekatA, ...ObjekatB};
```

Ovde smo stali

Komponovanje agregacijom

- Agregacije su odlične za primenu univerzalnih apstrakcija, kao što je primena funkcije na svaki član agregata (npr. `array.map(fn)`), transformacija vektora kao pojedinačnih vrednosti, itd. Može se implementirati na širokom dijapazonu struktura (nizovi, mape, stabla, grafovi, skupovi).
- Međutim, ako postoji potencijalno stotine hiljada ili milioni podobjekata, obrada strima ili delegiranje su efikasniji.
- Slede primeri dve agregacije: agregacija niza i agregacija povezane liste po parovima. Za oba primera ulazni podaci su isti:

```
const objs = [
 { a: 'a', b: 'ab' },
 { b: 'b' },
 { c: 'c', b: 'cb' }
 ];
```

Agregacija niza 1/2

```
// Agregacija niza

function kolekcija(a, e) { return a.concat([e])};

IspisiNaslov ('Ulazni objekat')

IspisiObjekat (objs)// [{"a": "a","b": "ab"},{"b": "b"},{"c": "c", "b": "cb"}]


 // Agregacija niza

const nizA = objs.reduce(kolekcija, []);

IspisiNaslov ('Agregacija kolekcije')

IspisiObjekat (nizA)   // [{"a": "a","b": "ab"},{"b": "b"},{"c": "c", "b":
"cb"}]

console.log(`nabrojivi ključevi: ${ Object.keys(nizA) }`) // 0, 1, 2

console.log('element nizA[0].b: ’, nizA[0].b, 'element nizA[1].b: ',
nizA[1].b,' element nizA[2].c: ', nizA[2].c)
```

Agregacija niza 2/2

```
Ulazni objekat
[  {    "a": "a",    "b": "ab"  },
   {    "b": "b"  },
   {    "c": "c",    "b": "cb"  }]
```

Agregacija kolekcije

```
[ {  "a": "a",  "b": "ab" },
  {  "b": "b" },
  {  "c": "c",  "b": "cb" }]
```

nabrojivi ključevi: 0,1,2

element **nizA[0].b: ab** element **nizA[1].b:b** element **nizA[2].c:c**

Agregacija povezane liste po parovima

```
// Agregacija povezane liste

function par (a, b) { return [b, a]};


IspisiNaslov ('Ulazni objekti')

IspisiObjekat (objs) // [{"a": "a","b": "ab"},{"b": "b"},{"c": "c", "b": "cb"}]


const povezanaLista = objs.reduceRight(par, []);

IspisiNaslov ('Agregacija povezane liste po parovima')

IspisiObjekat (povezanaLista)/* [{"a": "a","b": "ab"},[{"b": "b"},[{"c": "c","b":
"cb"}, []]]] */

console.log(`nabrojivi ključevi: ${ Object.keys(povezanaLista) }`) // 0, 1

for (const [key, value] of Object.entries(povezanaLista)) {

  console.log(`${key}: ${value}`);

}

console.log('Povezana lista: ', povezanaLista) //
```

Komponovanje delegiranjem

- Delegiranje je kada objekat prosleđuje ili delegira drugom objektu. Omogućuje uštedu memorije i dinamičko ažuriranje velikog broja instanci.
  - Štednja memorije: U svakom trenutku može postojati potencijalno mnogo instanci objekta pa bi bilo korisno deliti identična svojstva ili metode među instancama i time smanjiti memorijske zahteve.
  - Dinamičko ažuriranje više instanci: Svaki put kada više instanci objekta treba da dele identično stanje koje će možda morati da se ažurira dinamički i da se promene trenutno odražavaju u svakoj instanci.

Primer: agregacija delegiranjem

```
// Agregacija delegiranjem

const delegiraj = function (a, b) { return Object.assign(Object.create(a), b)};

IspisiNaslov ('Ulazni objekat')
IspisiObjekat (objs) //[{"a": "a","b": "ab"},{"b": "b"},{"c": "c", "b": "cb"}]

const d = objs.reduceRight(delegiraj, {});

IspisiNaslov ('Agregacija delegiranjem')
IspisiObjekat (d) //{"a": "a",  "b": "ab"}
console.log(`nabrojivi ključevi: ${ Object.keys(d) }`) // a, b
console.log('d.b: ', d.b,' d.c: ', d.c) // d.b:ab  d.c:c
```

Komponovanje objekata: klasni obrazac

- Jedan od najrasprostranjenijih načina komponovanja je **klasni obrazac dizajna**.
- the Gang of Four, “Design Patterns: Elements of Reusable Object Oriented Software”, preporuka: “Favor object composition over class inheritance”

Zašto ta preporuka?

```
//komponovanje klasnim obrascem
class Foo {
   constructor () {
   this.a = 'a'
   }
}
class Bar extends Foo {
   constructor (options) {
   super(options);
      this.b = 'b'
   }
}
const myBar = new Bar(); // Rezultat: {a: 'a', b: ’b’}

// Isto to urađeno konkatenacijom
const a = {
   a: 'a'
};
const b = {
   b: 'b'
};
const c = {...a, ...b}; // Rezultat : {a: 'a', b: 'b'}
```

Nedostaci klasnog obrasca

- **Problem tesnog sprezanja:** Zbog zavisnosti naslednickih klasa (dece) od implementacije roditeljske klase, klasno nasleđivanje je najčvršće sprezanje u objektno orijentisanom dizajnu.
- **Problem fragilne bazne klase:** Zbog tesnog sprezanja, promene u baznoj klasi mogu potencijalno da da izazovu poremećaje u velikom broju naslednickih klasa – što je posebno nezgdno ako su one pod kontrolom treće strane.
- **Problem nefleksibilne hijerarhije:** Zbog taksonomije jednog pretka (super klasa), vremenom sve klasne taksonomije postaju neupotrebljive za nove slučajeve.
- **Problem dupliranja po potrebi:** Zbog nefleksibilnih hijerarhija, novi slučajevi se implementiraju dupliciranjem a ne proširivanjem što vodi novim klasama koje su neočekivano divergentne. Kada se jednom pribegne dupliciranju, nije očigledno od koga ili zašto nove klase treba da nasleđuju.
- **Gorila/banana problem: "…** problem sa objektno orijentisanim jezicima je što oni imaju svo to implicitno okruženje koje naokolo vuku sa sobom. Vi ste želeli bananu, ali je ono što ste dobili gorila koji drži bananu, i cela džungla okolo." *Joe Armstrong, "Coders at* Work"

FP: Apstrakcija koda$_{1/2}$

- ***Apstrakcija*** koda je izdvajanje funkcionalnosti u funkcije koje se mogu primenjivati svaki put kada odgovarajuća funkcionalnost treba da se ostvari – u stvari, pravljenje funkcija.
- Primer: Nad nizovima vrlo često određene operacije treba primeniti na sve elemente niza.
  - Apstrakcija za ovu funkcionalnost može da bude funkcija `forEach`

FP: Apstrakcija koda$_{2/2}$

```
const forEach = (arrayIn, fn) => {
// arrayIn        – originalni niz
// fn – funkcija koja se primenjuje na članove arrayIn
// arrayOut -  povratna vrednost; niz koji je rezultat
//primene funkcije fn

let i;

let arrayOut = []

   for(i=0; i<arrayIn.length; i++){

      arrayOut [i] = fn(arrayIn[i])

     }

return arrayOut

}


let ulaz = [1,2,3]

console.log (' Izlaz za fn => data: ', forEach(ulaz, data => data))

console.log (' Izlaz za fn => 2*data: ', forEach(ulaz, (data) => 2*data))
```

FP: Deklarativnost koda

- Deklarativni stil programiranja rezultuje kodom koji izvršiocu kaže ŠTA TREBA DA URADI da bi rešio određeni zadatak :

```
let ulazniNiz = ["jedan", "dva", "tri"]
let izlazniNiz = []

// Šta se radi – prepisuje se jedan niz u drugi niz u petlji
izlazniNiz = forEach(ulazniNiz, data => data)
IspisiObjekat(ulazniNiz)
IspisiObjekat(izlazniNiz)
```

Zašto je funkcionalno programiranje deklarativno

- Funkcionalno programiranje je stil programiranja koji za rešavanje zadataka koristi samo koncepte *funkcije* i *kompozicije funkcija*.
- Kompozicija funkcija je opet funkcija koja za argument ima neku funkciju (ne nužno drugu, može i istu - *rekurzija*).
- Dakle, svi programi ovde rade **istu stvar – izvršavaju funkcije**.
- Zbog toga nema razloga da se detaljno opisuje KAKO program radi, dovoljno je napisati ŠTA radi, odnosno koje funkcije izvršava.

Prednosti FP-a

- Oslonac na čvrstu teorijsku osnovu u matematici (Lambda račun i teorija kategorija) garantuje konzistentnost pa i robusnost jezicima funkcionalnog programiranja.
- Kod koji je lakši za testiranje zato što ima kontrolisane bočne efekte.
- Upotrebljiviji i razumljiviji (čitljiviji) kod – funkcija kao apstrakcija i kompozibilnost funkcije daju ponovno upotrebljiv kod a deklarativnost poboljšava čitljivost koda

```
// Imperativno
let dva = 2
let cetiri = 2*2
// a šta dalje da bismo dobili 8, ili 16, ili 32 …?

// Deklarativno (funkcionalno)
const double = (value) => value * 2
// a za 8, 16, 32 … jednostavno:
(double (double (2)));
(double (double (double(2))))
(double (double (double(double(2)))))
```

Nedostaci FP-a

- **“Suštinski”**
  - **Terminološki problemi** koji, delom, sigurno potiču iz čvrstog oslonca FP-a na matematiku uz istovremenu činjenicu da klijentela (prosečni programeri) nije baš mnogo familijarna sa matematikom.
  - **Ne-funkcionalna priroda današnjih računara** – računar koji mi koristimo je Turingova mašina, a ne Čerčov Lambda računar pa se funkcionalno programiranje ne može direktno implementirati do hardvera.
- **“Tehnički”**
  - **Ulaz/izlaz** : Savki U/I predstavlja bočni efekat pa je inherentno ne-funkcionalan.
  - **Cena rekurzije**: Rekurzija se smatra jednim od najboljih delova FP-a, ali joj je cena (memorijski i procesno) veoma visoka .
  - **Problem programiranja stanja:** Predstavljanje stanja u funkcionalnom programiranju nije ni blizu intuitivno kao većina drugih funkcionalnih operacija

Ublažavanje nedostataka FP-a

- **“Suštinski”**
  - **Terminološki**: FP se sa ovim problemom nosi tako što prosečan korisnik operiše sa pojednostavljenim konceptima i idejama svedenim na direktne praktične primene.
  - **Ne-funkcionalna priroda današnjih računara :** FP se sa ovim problemom bori tako što realno sagledava nemogućnost primene modela “kornjače skroz do dole”.
- **“Tehnički”**
  - **Ulaz/izlaz** : FP se sa ovim problemom bori tako što vrši izolovanje U/I-a.
  - **Cena rekurzije**: FP se sa ovim problemom bori tako što uvodi partikularna rešenja (n.pr. terminalna rekurzija).
  - **Problem programiranja stanja:** FP se sa ovim problemom nosi razvojem posebnih metoda za funkcionalnu reprezentaciju stanja i pratećih softverskih alata kao što je, na primer, `Redux.js`.

Sažetak 1/4

- **Tipovi i sistemi tipiziranja**
  - U matematici, logici i računarskoj nauci sistem tipova je formalni sistem u kome je svakom terminu dodeljen "tip" koji definiše njegovo značenje i operacije koje se nad njim mogu izvršiti.
  - U praktičnom računarstvu - programiranju - tipovi su meta-jezik koji pomaže da se izbegnu greške u programima.
  - U praksi se koriste različiti sistemi tipiziranja od kojih je (posebno u jezicima FP-a) vrlo zastupljen sistem koji se zove "Hindley-Milner" .

Sažetak 2/4

- **Apstrakcija**
  - **Proces** uklanjanja fizičkih, prostornih ili vremenskih detalja ili atributa u proučavanju objekata ili sistemima da bi se pažnja usmerila na detalje veće važnosti.
    - Ima dve glavne komponente **generalizaciju** i **specializaciju**.
    - Rezultat procesa su **apstraktni koncepti-objekti** koji odslikavaju zajednička svojstava ili atributa različitih ne-apstraktnih objekata ili sistema.
    - Može da DOPRINESE I POJEDNOSTAVLJENJU PROBLEMA tako što će se problem sagledati kroz proces izdvajanja suštine i koncepata.
    - Osnova za implementaciju jednog od osnovnih principa arhitekture softvera - PONOVNE UPOTREBLJIVOSTI (engl. reusability)
- **Funkcije** su pogodne za apstrakciju zato što poseduju dve odlike od suštinskog značaja za dobru apstrakciju:
  - **Identitet** – Mogućnost dodele imena i ponovno korišćenje u različitim kontekstima.
  - **Kompozabilnost** – Mogućnost da se od jednostavnijih funkcija komponuju složenije funkcije.

Sažetak 3/4

- Funkcionalno programiranje
  - Funkcionalno programiranje (FP) je stil programiranja koji komponuje aplikacije korišćenjem čistih funkcija izbegavajući mutabilna stanja i bočne efekte.
  - U funkcionalnom programiranju komponente su funkcije a mehanizam kompozicije je funkcija višeg reda.
- Osnovni koncepti funkcionalnog programiranja su:
  - Funkcija
  - Nepromenljivost (imutabilnost)
  - Komponovanje
  - Deklarativno izražavanje

Sažetak$_{4/4}$

- Prednosti FP-a
  - Oslonac na čvrstu teorijsku osnovu u matematici (Lambda račun i teorija kategorija).
  - Kod koji je lakši za testiranje zato što ima kontrolisane bočne efekte.
  - Upotrebljiviji i razumljiviji (čitljiviji) kod .
- Nedostaci FP-a
  - “Suštinski”
    - Terminološki problemi.
    - Ne-funkcionalna priroda današnjih računara.
  - “Tehnički”
    - Ulaz/izlaz
    - Cena rekurzije
    - Problem programiranja stanja

Literatura za predavanje

1. Z. Konjović, Funkcionalno programiranje, **Tema 04 – Uvod u FP**, slajdovi sa predavanja, dostupni na folderu **FP kurs 2023-24→Files →Slajdovi sa predavanja**
2. Eric Elliot, **Composing Software - An Exploration of Functional Programming and Object Composition in JavaScript**, Leanpub, 2019,
3. Kyle Simpson, **Functional-Light JavaScript - Balanced, Pragmatic FP in JavaScript**, Manning, 2018