DEO III: FUNKCIONALNO PROGRAMIRANJE U JEZIKU JavaScript

Sadržaj

Poglavlje 8: FUNKCIONALNO PROGRAMIRANJE: ZAŠTO, KAKO?

U ovom poglavlju pokušaćemo da objasnimo kroz primere razloge zbog kojih je korisno razumeti i naučiti funkcionalno programiranje i da izložimo osnovne koncepte i principe funkcionalnog programiranja.

U knjizi [1], na koju smo puno oslonjeni, njen autor Birian Lonsdorf započinje poglavlje posvećeno kuriranju sledećom rečenicom: "Moj tata je jednom objasnio kako postoje određene stvari bez kojih čovek može da živi dok ih ne nabavi.„ Čini mi se da se ovo može odnositi na funkcionalno programiranje u celini. U nastavku ćemo krenuti da "nabavljamo" stvari koje čine funkcionalno programiranje. Smatraćemo da smo bili uspešni ako se nakon čitanja ove knjige upitate: kako sam ja bez ovoga programirao?

8.1. Zašto funkcionalno programiranje: primer jata galebova

Prvi primer koji ćemo pokazati je primer koji smo preuzeli iz izvora [1]. Primer predstavlja poređenje dva programa koji (treba da) reše isti problem. Jedan program napisan je korišćenjem objektne paradigme, a drugi korišćenjem funkcionalne paradigme. Programi su kodirani sa ciljem da se drastično pokažu prednosti funkcionalnog programiranja pa je, zbog toga, objektno rešenje u priličnoj meri karikaturalno. Ipak, iz primera se jasno može uočiti koliko je lakše razumeti kod pisan funkcionalnim stilom od koda pisanog objektnim stilom što je i bio cilj. Pa da pogledamo taj primer.

Galebovi su ptice koje žive u kolonijama – jatima i razmnožavaju međusobno se pareći. Kada se jata galebova spajaju, formiraju se veća jata a kada se galebovi razmnožavaju jato se povećava za broj parenja galebova. Pretpostavimo da imamo tri kolonije **JatoA**, **JatoB** i **JatoC**, svaku sa određenim brojem jedinki i da nam je cilj da odredimo broj jedinki nakon spajanja jata i razmnožavanja (u jednoj generaciji).

Primer koda pisanog objektnim stilom sledi:

```
class Jato {
      constructor(n) {
            this.galebovi = n;
 }
 spajanje(drugoJato) {
      this.galebovi += drugoJato.galebovi;
            return this;
 }
 razmnozavanje(drugoJato) {
      this.galebovi = this.galebovi * drugoJato.galebovi;
            return this;
 }
}
const jatoA = new Jato(4);
const jatoB = new Jato(2);
const jatoC = new Jato(0);

IspisiNaslov (' Objekti pre poziva metoda \n')
```

```
IspisiNaslov (' Objekat jatoA \n')
IspisiObjekat (jatoA);
IspisiNaslov (' Objekat jatoB \n')
IspisiObjekat (jatoB);
IspisiNaslov (' Objekat jatoC \n')
IspisiObjekat (jatoC);

/* Izvršavanje ide obrnutim redosledom od redosleda navođenja metoda;
međurezultati se smeštaju u jatoA.galebovi */
const rezultat = jatoA
      .spajanje(jatoC)                       //jatoA.galebovi=32+0=32   (4)
      .razmnozavanje(jatoB)                  //jatoA.galebovi=16*2=32   (3)
      .spajanje(jatoA.razmnozavanje(jatoB))  //jatoA.galebovi=8+8=16    (2)
      .galebovi                              //jatoA.galebovi=4         (1)

IspisiNaslov (' Objekti nakon poziva metoda \n')

IspisiNaslov (' Objekat jatoA \n')
IspisiObjekat (jatoA);
IspisiNaslov (' Objekat jatoB \n')
IspisiObjekat (jatoB);
IspisiNaslov (' Objekat jatoC \n')
IspisiObjekat (jatoC);

console.log('Broj galebova nakon spajanja jata i razmnožavanja: ',
jatoA.galebovi)
```

Razultat izvršvanja ovoga koda je sledeći ispis:

Objekti pre poziva metoda

Objekat jatoA

```
{
  "galebovi": 4
}
```

Objekat jatoB

```
{
  "galebovi": 2
}
```

Objekat jatoC

```
{
  "galebovi": 0
}
```

Objekti nakon poziva metoda

Objekat jatoA

```
{
  "galebovi": 32
}
```

Objekat jatoB

```
{
  "galebovi": 2
}
```

Objekat jatoC

```
{
  "galebovi": 0
}
```

Broj galebova nakon spajanja jata i razmnožavanja: 32

Po „zdravoj logici“ očekivali smo da konačan rezultat bude 16 = 0+4*2+8+0. Ali nije, jer ovaj program radi ono što smo u njemu objektno kodirali, a ne ono što smo mislili da smo kodirali. Dešava se sledeće.

Pri kreiranju objekata `jatoA`, `jatoB` i `jatoC` svim objektima je kreirano svojstvo sa ključem `galebovi` i vrednostima 4, 2, 0 respektivno. Pozivi metoda izvršavaju se “iznutra” što znači od poslednje navedene metode ka prvoj.

Da vidmo kako zadatak može da se reši sa malo funkcionalnog pristupa. Kod može da izgleda ovako:

```
const spajanje = (jatoX, jatoY) => jatoX + jatoY;
const razmnozavanje = (jatoX, jatoY) => jatoX * jatoY;
      const jatoA = 4;
      const jatoB = 2;
      const jatoC = 0;
      const rezultat =
      spajanje(razmnozavanje(jatoB, spajanje(jatoA, jatoC)),
      razmnozavanje(jatoA, jatoB));
      console.log('Broj galebova nakon spajanja jata i razmnožavanja: ',
                  rezultat)
```

Ovo je kod koji se sigurno lakše čita i razume nego prethodni. A i rezultat je ono što smo očekivali:

Broj galebova nakon spajanja jata i razmnožavanja: 16

I tu nije kraj. Sa funkcionalnim pristupom još svašta korisno može da se uradi.

Ako se malo bolje pogledaju funkcije `spajanje()` i `razmnozavanje ()` nije teško zaključiti da se tu radi o prostom sabiranju, odnosno množenju dva broja. To znači da će sledeći kod da radi istu stvar:

```
const saberi = (x, y) => x + y;
const pomnozi = (x, y) => x * y;
      const jatoA = 4;
      const jatoB = 2;
      const jatoC = 0;
      const rezultat =
      saberi (pomnozi (jatoB, saberi (jatoA, jatoC)), pomnozi (jatoA,
      jatoB));
      console.log('Broj galebova nakon spajanja jata i razmnožavanja: ',
                  rezultat)
```

Dakle, funkcije `saberi()` i `pomnozi()` mogu da se napišu jednom i da se koriste svaki put kada su nam potrebne – funkcionalno programiranje je pogodno za apstrakciju.

Podsetimo se, sada, osnovnih aritmetičkih zakona da bismo kasnije videli kako možemo da ih iskoristimo da poboljšamo naš kod. Predstavićemo ih i njihovo važenje ćemo da demonstriramo odgovarajućim kodom u JavaScript-u:

```
// Važe zakoni aritmetike
const saberi = (x, y) => x + y;
const pomnozi = (x, y) => x * y;
let x = 2, y = 3, z = 4;

// asocijativnost
```

```
   let asocijativno = saberi(saberi(x, y), z) === saberi(x, saberi(y, z))
   console.log('Asocijativnost važi: ',asocijativno) // true

// komutativnost
let komutativno = saberi(x, y) === saberi(y, x);
console.log('Komutativnost važi: ',komutativno) // true

// identitet
let identitet = saberi(x, 0) === x;
console.log('Identitet važi: ',identitet) // true

// distributivnost
let distributivno = pomnozi(x, saberi(y,z)) === saberi(pomnozi(x, y),
                                                       pomnozi(x, z));
console.log('Distributivnost važi: ',distributivno) // true
```

Iskoristićemo ih da malo doteramo sledeću liniju koda koja baš nije šampion čitljivosti:

```
saberi (pomnozi (jatoB, saberi (jatoA, jatoC)), pomnozi (jatoA, jatoB));
```

Znajući da `jatoC` nema ni jednu pticu, možemo da primenimo zakon identiteta

`(saberi (jatoA, jatoC) === jatoA )`

pa ćemo da dobijemo ekvivalentan kod

```
saberi (pomnozi (jatoB, jatoA), pomnozi (jatoA, jatoB));
```

Ako sada na ovu liniju primenimo zakon distributivnosti, dobijamo sledeći ekvivalentan kod:

```
pomnozi(jatoB, saberi(jatoA, jatoA));
```

Rezultat ovih intervencija na kodu čiju nam ispravnost garantuju zakoni matematike (a nema jačeg garanta od matematike da je nešto ispravno) je sledeći kod:

```
const saberi = (x, y) => x + y;
const pomnozi = (x, y) => x * y;
      const jatoA = 4;
      const jatoB = 2;
      const jatoC = 0;
      const rezultat = pomnozi(jatoB, saberi(jatoA, jatoA));
      console.log('Broj galebova nakon spajanja jata i razmnožavanja: ',
                  rezultat)
```

Rezultat koji se dobije izvršavanjem ovoga koda je korektan, a kod je malo čitljiviji. Pored bolje čitljivosti, ovaj novi kod se odlikuje još jednim, u realnim uslovima izuzetno važnim svojstvom, manjim brojem izvršavanja funkcija. U polaznom kodu imali smo **dva poziva** funkcije `saberi()` i **dva poziva** funkcije `pomnozi()` a u novom kodu imamo **po jedan** poziv svake funkcije. Manji broj pozivanja funkcija znači kraće vreme izvršavanja programa. To nije značajno ako se naš kod izvršava nekoliko desetina ili nekoliko stotina puta, ali postaje značajno ako se on izvršava nekoliko miliona puta.

Loša vest je ovde da je naš primer naivan i da u realnim aplikacijama aparat funkcionalnog programiranja koji implementira elementarnu matematiku, poput aritmetičkih zakona, ne može mnogo da nam pomogne.

Ali postoji i dobra vest: funkcionalno programiranje poseduje bogatu aparaturu opet oslonjenu na matematiku (teorija skupova, lambda račun i teorija kategorija) koja omogućuje da se prave elegantni, čitljivi i performantni programi.

A dobra vest je i da ćemo matematiku u ovoj knjizi posmatrati strogo u funkciji programiranja. Isplativost principijelnog matematičkog okvir za programiranje je zapanjujuća i u ovoj knjizi pokušaćemo do to i pokažemo. Nadamo se da ćemo time pomoći da, ukoliko ga već nemate, uspostavite "saradnički", možda čak i "prijateljski" odnos sa matematikom.

Naravno, kao "programeri praktičari" sigurno ćete se upitati: Zašto moram da razumem matematički aparat koji je u pozadini, ako već u programskim jezicima posedujem njegovu implementaciju?

Odgovor je u sledećoj alegoriji: pecaroš koji zna šta smuđ voli da jede, možda će i da ga upeca. Pecaroš koji to ne zna, može doveka da peca sa pogrešnim mamcem; možda upeca šarana, ali **smuđa nema**. Tako vam je i u programiranju: da ne biste trošili vreme i snagu na pravljenje neodgovarajućih programa, što u našoj maloj alegoriji odgovara upecanom šaranu, treba da posedujete dublje znanje o jeziku u kome programirate. A dublje znanje o jezicima funkcionalnog programiranja je u granama matematike koje smo pomenuli.

8.2. Kako funkcionalno programiranje: osnovni principi/ koncepti

Funkcionalno programiranje je programska paradigma koja ima svoje korene u matematici, prvenstveno iz lambda računa. Funkcionalno programiranje ima za cilj da bude deklarativno i tretira aplikacije kao rezultat kompozicije čistih funkcija. Na taj način se izbegavaju problemi koji dolaze sa deljenim stanjem, promenljivim podacima i neželjenim efektima koji su uobičajeni u objektno orijentisanom programiranju i drugim stilovima programiranja.

U nastavku ćemo izložiti osnovne principe/koncepte koji se primenjuju u funkcionalnom programiranju.

8.2.1. Odsustvo (kontrola) bočnih efekata

Bočni efekti su stvari kao što su operacije ulaza/izlaza, različita logovanja, upozoravajuće i terminirajuće greške, mrežni pozivi i izmena spoljne strukture podataka ili promenljive. U suštini sve što čini sistem nepredvidivim. Iako funkcionalno programiranje u teoriji zagovara odsustvo bočnih efekata, u praksi je zastupljen princip kontrole bočnih efekata – bočni efekti dozvoljeni su ako su predviđeni a krajnji cilj je da se smanji pojava neželjenih i nepredvidivih bočnih efekata koliko god je to moguće.

8.2.2. Referencijalna transparentnost

Princip referencijalne transparentnosti zahteva da funkcija za iste vrednosti argumenata uvek vraća isti rezultat. Na taj način se obezbeđuje da se zamenom funkcije njenom povratnom vrednošću neće promenutu ponašanje programa.

8.2.3. Čista funkcija

Čista funkcija je funkcija koja je referencijalno transparentna (za isti ulaz uvek proizvodi isti izlaz) i nema bočnih efekata.

8.2.4. Imutabilnost

Nepromenljivost je u osnovi funkcionalnog programiranja. Nepromenljivost je ideja da kada se vrednost jednom deklariše, ona je nepromenljiva i na taj način čini ponašanje u programu mnogo predvidljivijim.

8.2.5. Funkcije kao entiteti prve klase

Ovaj princip obezbeđuje da se sa funkcijama može raditi sve ono što se može raditi i sa ostalim tipovima jezika. To znači da se funkcije mogu dodeljivati varijablama i skladištiti u strukture podataka.

8.2.6. Funkcije višeg reda

Fukcije višeg reda su funkcije koje mogu da primaju druge funkcije kao argumente i da vraćaju funkciju kao izlaznu vrednost. U funkcionalnom programiranju one su osnova bazčnog mehnaizma apstrakcije koji se zove kompozicija funkcija.

8.2.7. Disciplinovano stanje

Sve što se dešava pri izvršavanju računarskog programa čuva se u memoriji koja se naziva **stanje programa**. Za razliku od **deljenog stanja**, gde je stanje zajedničko za ceo program i svako izvršavanje ima neograničen pristup njegovom sadržaju i za čitanje i za pisanje, **disciplinovano stanje** implementra mehanizme kojima se kontroliše čitanje i pisanje, odnosno kojima se ograničava pravo izvršavanjima da pristupaju određenim delovima stanja.

8.2.8. Sistemi tipova

Korišćenjem tipiziranja i sistema tipova onemogućuje se ”mešanje baba i žaba ” – u tipiziranim sistemima kompajler/interpreter može da otkrije situacije u kojima program sadrži stvari koje nisu dozvoljene. Na primer, da upozori putem greške na nedozvoljenu radnju aritmetičkog sabiranja vrednosti koje nisu numeričkog tipa pre no što dođe do izvršenja koda.

8.2.9. Preferiranje rekurzije u odnosu na petlju

Rekurzija („samopozivanje“ funkcije) se često koristi umesto petlji u funkcionalnom programiranju. Razlog tome je što se na taj način dosledno primenjuje kompozicija funkcija kao osnovni mehanizam apstrakcije u kodu - funkcija poziva samu sebe da ponavlja operacije dok se ne dostigne bezni slučaj rekurzije.

Literatura uz Poglavlje 8

1. Brian Lonsdorf, Professor Frisby's Mostly Adequate Guide to Functional Programming, Samizdat, 2019

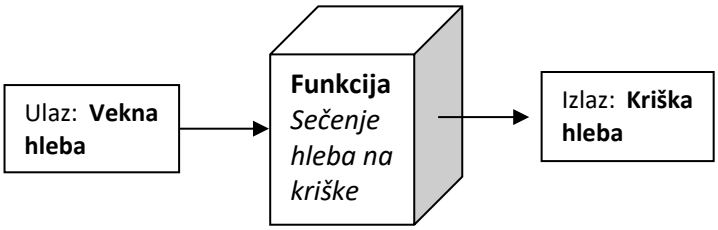
Poglavlje 9: FUNKCIJA JE CENTRALNA U FUNKCIONALNOM PROGRAMIRANJU

Koncept funkcije prisutan je u nekom obliku praktično u svim programskim paradigmama i svim programskim jezicima sa kojima se susreće prosečan programer (u programere ne računamo one koji „programiraju“ samo u markerskim jezicima) .

Naravno, u funkcionalnom programiranju igra posebnu ulogu i ima poseban status.

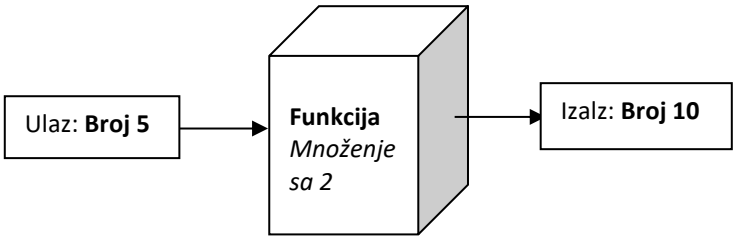
9.1. Uloga funkcije u funkcionalnom programiranju

U funkcionalnom programiranju funkcija je osnovni konstrukt programskog jezika i svi drugi konstrukti se realizuju putem funkcija. Mehanizam funkcije sastoji se u sledećem: funkcija prima neki ulaz, nešto uradi sa tim ulazom da proizvede rezultat, i vraća taj rezultat kao izlaz. Mogli bismo to da vizuelno ilustrujemo slikom 5.1.



Slika 5.1 Funkcija sečenja hleba na kriške

Suštinski isto se dešava i kada vršimo neko računanje, recimo množimo sa 2. Zato je i slika 5.2 u osnovi ista kao slika 5.1.



Slika 5.2 Funkcija množenje sa 2

U funkcionalnom programiranju SVE SE RADI pomoću funkcija. Pa kako onda uspevamo da uradimo sve što nam treba a imamo samo jedan alat? Odgovor je, naravno, u matematici i zove se Lambda račun (λ-račun).

Teoriju Lambda računa razvio je američki matematičar Alonzo Čerč (Alonzo Church) tridesetih godina prošlog veka izučavajući matematička svojstva efektivno sračunljivih funkcija (sračunljive funkcija su one funkcije čije vrednosti mogu da se sračunaju pomoću nekog algoritma). Pokazalo se da je taj rezultat mnogo vredniji – ono što je Alonzo Čerč uradio postalo je fundament funkcionalnog programiranja, odnosno funkcionalnih programskih jezika.

O λ-račun u ovoj knizi postoji posebno poglavlje a ovde ćemo da kažemo samo jednu stvar koja nam je važna da bismo razumeli dva koncepta funkcionalnog programiranja, ***parcijalnu aplikaciju funkcija*** i ***kuriranje***. To je sledeća stvar. Baš kao i na prethodnim slikama, funkcija u λ-računu **prima jedan ulaz** i **vraća jedan izlaz**.

9.1.1. Funkcije su “građani prve klase”

Kada se za neki programski jezik kaže da tretira funkcije kao „građane prve klase“, ne misli se ništa što bi funkciju pozicioniralo u kontekst klasnog objektnog programiranja. Misli se, prosto, da su funkcije kao i svi ostali konstrukti u programskom jeziku, drugim rečima NORMALAN TIP.

To znači da se funkcije mogu tretirati kao i svi drugi tipovi u jeziku - mogu se dodeljivati promenljivim, skladištiti u nizovima i objektima, prosleđivati kao parametri funkcije, vraćati kao izlaz funkcije. Jedina specifičnost funkcija u odnosu na ostale konstrukte jezika je da se one ***mogu pozivati*** i nema ništa drugo posebno u vezi sa njima. Za ilustraciju smo pozajmili odličan primer koji je dao Majkl Fogus (Michael Fogus) u svojoj knjizi [Michael Fogus, Functional JavaScript]:

- Numerička vrednost može se uskladištiti u varijablu, a može i u funkciju:
```
let fortytwo = 42;
fortytwo = function() { return 42 };
```
- Numerička vrednost može se uskladištiti u element niza, a može i funkcija:
```
let fortytwos = [42, function() { return 42 }];
```
- Numerička vrednost može se uskladištiti u svojstvo objekta, a može i funkcija:
```
let fortytwos = {number: 42, fun: function() { return 42 }};
```
- Numerička vrednost može se navesti u izrazu, a može i funkcija:
```
42 + (function() { return 42 })();
//=> 84
```
- Numerička vrednost se može proslediti funkciji, a može joj se proslediti i funkcija:
```
function weirdAdd(n, f) { return n + f() }
weirdAdd(42, function() { return 42 });
//=> 84
```
- Funkcija može da vrati numeričku vrednost, a može da vrati i funkciju:
```
function fun1 () { return 42 };
console.log(fun1()) // Ispis: 42
function fun2 () {return function() { return 42 }};
console.log(fun2()) // Ispis: 42
```

Međutim, i vrlo površno pregledanje koda na github-u otkriva kolektivno izbegavanje ove činjenice što je, najverovatnije, posledica široko rasprostranjenog nerazumevanja pravog koncepta funkcije. Evo opet jednog karikaturalnog primera kojim Brajan Loson u svojoj knjizi [professor Frisby] ilustruje situaciju:

```
const zdravo = ime => `Zdravo ${ime}`;
const pozdrav = ime => zdravo(ime);
```

U ovom primeru imamo kao potpun višak omotačku funkciju `pozdrav` oko funkcije `zdravo`. Zašto bi to bio višak? Zato što su funkcije u JavaScript-u samo posebna vrsta objekta koji se *može pozvati*. Par malih zagrada `()` iza imena funkcije znači pozivanje funkcije. Ako malih zagrada nema, biće vraćen izvorni kod funkcije. Ovde je `zdravo` već funkcija koja prima jedan argument i može se pozvati direktno, pa nema nikakvog razloga da se ona „pakuje“ u drugu funkciju u kojoj se samo vrši njeno pozivanje. Evo primera koji to potvrđuje:

```
console.log(zdravo('Pero'))   // Ispis: Zdravo Pero
console.log(pozdrav('Pero'))  // Ispis: Zdravo Pero
```

Ako baš želite da funkciju `zdravo()` pozovete u nekoj drugoj funkciji, možete i da je prosledite kao argument:

```
const zdravo = ime => `Zdravo ${ime}`;
```

```
const JaBihDaPozdravim =  (pozdravnaFunkcija, koga, sta) =>
                          {console.log (pozdravnaFunkcija(koga));
                           console.log (' poslao si mi ', sta);
}

JaBihDaPozdravim (zdravo, 'Pero', ', baš lepo pismo');
```

Omotavanje funkcija je dosta raširena, ne uvek dobra praksa za odlaganje evaluacije funkcije. U pitanju je postavljanje poziva željene funkcije u drugu funkciju koja će, kada bude pozvana, izvršiti poziv željene funkcije. Ovakva praksa može da ima ozbiljne posledice na održavanje koda. U knjizi [Loson] ćete naći još primera sklonosti ka dodavanju slojeva indirekcije koji ne donose nikakvu vrednost, nego samo povećavaju količinu redundantnog koda koji treba pregledati i održavati. Sledeći primer iz [Loson] lepo ilustruje kako nekorišćenje/ korišćenje činjenice da su funkcije "građani prve klase" može da vam zagorča/olakša održavanje koda.

Na primer liniji koda

```
httpGet('/post/2', json => renderPost(json));
```

nedostaje rukovanje greškom. Jedan način uključivanja rukovanja greškom koji se odmah nameće je da se Ako bi se svaki `httpGet` poziv u kodu i popravi da eksplicitno prosledi grešku (dodaje mu se argument i u definiciji i u pozivu):

```
/* Sada lepo idi natrag na svaki httpGet poziv u kodu i popravi ga da
eksplicitno prosledi grešku (dodaj mu argument i u definiciji i u pozivu).
*/
httpGet('/post/2', (json, err) => renderPost(json, err));
```

Gotovo je sigurno da bi se na ovaj način napravila greška u kodu koji ima više poziva.

Ako bi se iskoristilo znanje o funkcijama u vezi sa prosleđivanjem argumenata, to bi moglo da se uradi i ovako:

```
// renderPost se poziva iz httpGet sa onoliko argumenata koliko želi
httpGet('/post/2', renderPost);
```

Osim uklanjanja nepotrebnih funkcija, tu je i pitanje imenovanja i referenciranja argumenata. Imena su prilično ozbiljan problem u praksi. Verovatnoća pojave potencijalno pogrešnih naziva raste sa širenjem kodne baze i izmenama zahteva. Postojanje više imena za istu stvar je uobičajen izvor zabune u projektima.

Tu je i pitanje generičkog koda. Na primer, sledeće dve funkcije rade potpuno istu stvar, ali je druga mnogo opštija i pogodnija za višekratnu upotrebu iako je samo imenovanje različito:

```
// Specifično za blog projekat koji se upravo radi
const validArticles = articles =>
articles.filter(article => article !== null && article !== undefined),

// Mnogo zgodnija za buduće projekte
const compact = xs => xs.filter(x => x !== null && x !== undefined);
```

I jedna i druga funkcija rade istu stvar – filtriraju stavke sa vrednošću različitom od `null` i `undefined`. Međutim, imenovanje u prvom slučaju prosto ne naglašava tu činjenicu – tesno je vezano je za termin Article pa čovek treba malo da se potrudi da uoči da je može koristiti u vrlo velikom broju situacija.

I na kraju još nešto ne baš tako nevažno – ključna reč `this`. Ako osnovna funkcija koristi ključnu reč `this` i mi to zovemo prvoklasnim, gotovo sigurno ćemo platiti ovaj previd u apstrakciji. Posebno ako nismo potpuno familijarni sa pravilima postavljanja ključne reči `this` u JavaScript-u koja nisu

jednostavna, a ne moraju uvek da izgledaju ni logično. Generalni je stav da u funkcionalnom programiranju treba izbegavati ključnu reč `this`, ali to nije uvek moguće – bilo je programera i programiranja puno ranije pa ima i raznih biblioteka koje se koriste. Ali da ipak citiramo Brajana Losona koji kaže: " Kao prljavu pelenu izbegavam da koristim `this`."

9.1.2. Parcijalna aplikacija i kuriranje funkcije

Parcijalna aplikacija i kuriranje funkcija su fundamentalne tehnike funkcionalnog programiranja. Zbog toga ćemo ih ovde detaljno objasniti.

I parcijalna aplikacija i kuriranje odnose se na izvršavanje funkcije koja ima više parametara, odnosno argumenata što je "prirodna potreba" svakog programiranja. Sa druge strane, rekli smo da je λ-računu funkcija prima jedan argument i vraća jedan izlaz. Dakle, potreban je mehanizam koji će omogućiti da se funkciji na neki način saopšti više argumenata i da se dobije izlaz sračunat na osnovu svih saopštenih argumenata. Mehanizam koji nam to omogućuje svodi se na ugnježdene funkcije.

Da bi se mogli implementirati ovi mehanizmi, jezik mora da podržava funkcije višeg reda. U JavaScript-u se za implementaciju ovih mehanizama koristi **zatvaranje**.

Najlakše ćemo to objasniti ako simuliramo "ručno" izvršavanje funkcije na primeru.

9.1.2.1. Parcijalna aplikacija funkcije

Posmatrajmo jednostavnu funkciju tri promenljive $f(x, y, z) = x + y + z$ pri čemu je oblast definisanosti funkcije $f(x, y, z)$ skup celih brojeva uključujući i 0. Vrednost funkcije za zadate vrednosti promenljivih x, y i z određuje se tako što se zadate vrednosti promenljivih uvrste u definiciju funkcije koja kaže da te tri vrednosti treba sabrati. Recimo da hoćemo da odredimo vrednost funkcije za $x = 1$, $y = 2$ i $z = 3$. To možemo da uradimo na različite načine.

Prvi način je da sve vrednosti promenljivih "odjednom" uvrstimo u formulu i izračunamo vrednost funkcije. To bi izgledalo ovako:

$$f(x = 1, \quad y = 2, \quad z = 3) = 1 + 2 + 3 = 6$$

Drugi načni može da bude da u prvom koraku u funkciju uvrstimo vrednosti za promenljive x i y a da vrednost z ne uvrstimo. Na taj način kao rezultat dobijamo novu funkciju koja zavisi od jedne promenljive z. Označimo je sa $g(z)$. To bi izgledalo ovako:

$$f(x = 1, \quad y = 2, \quad z) \; = \; 1 + 2 + z \; = \; 3 + z \; = g(z)$$

Ako u drugom koraku u funkciju $g(z)$ uvrstimo vrednost $z = 3$, dobićemo traženu vrednost funkcije $f(1, 2, 3)$:

$$g(z = 3) \; = \; 3 + 3 = 6 = f(x = 1, \quad y = 2, \quad z = 3)$$

Pri ovom načinu određivanja vrednosti funkcije nisu sve vrednosti promenljivih (argumenata) prosleđene funkciji odjednom da bi ona izračunala svoju vrednost . Umesto toga, desilo se sledeće:

1. Funkciji $f(x, y, z)$ prosleđene su vrednosti parametara x i y.
2. Funkcija $f(x, y, z)$ je, na osnovu prosleđenih vrednosti parametara $x = 1$ i $y = 2$ kao rezultat primene izraza koji je definiše (a to je izraz $x + y + z$) "sračunala" rezultat koje je nova funkcija $g(z)$ koja zavisi od argumenta z koji funkciji $f(x, y, z)$ nije bio raspoloživ u trenutku primene. U stvari, ova funkcija je u svom leksičkom okruženju fiksirala vrednosti prosleđenih argumenata $x = 1$ i $y = 2$.
3. Konačno, vrednost argumenta $z = 3$ prosleđen je novoj funkciji $g(z)$ i ona je "sračunala" svoj rezultat koje je, u stvari, rezultat koji se dobija kada se funkciji $f(x, y, z)$ proslede svi argumenti. Funkcija $g(z)$ je mogla to da uradi zato što je ona **zatvaranje koje ima pristup leksičkom okruženju funkcije iz prethodnog** koraka gde će da nađe vrednosti $x = 1$ i $y = 2$.

Na isti način mogli smo funkciji $f(x,y,z)$ da prosledimo u prvom koraku argumente x i z a u drugom koraku novoj funkciji (to bi sada bila funkcija $g(y)$) da prosledimo vrednost $y=2$ koja bi vratila rezultat $4+2=6$ što je i rezultat funkcije f. Takođe, mogli smo funkciji $f(x,y,z)$ da prosledimo u prvom koraku argumente y i z a u drugom koraku novoj funkciji (to bi sada bila funkcija $g(x)$) da prosledimo vrednost $x=1$ koja bi vratila rezultat $1+2+3=6$ što je, opet, i rezultat funkcije f.

Ovakav način određivanja vrednosti funkcije gde se funkciji u jednom koraku ne prosleđuju svi argumenti nego samo deo argumenata naziva se ***parcijalna aplikacija funkcije***.

U računarskim programima česte su ovakve situacije kada nisu poznate sve vrednosti argumenata, već samo vrednosti dela argumenata funkcije a to su situacije u kojima parcijalna aplikacija funkcija predstavlja rešenje problema.

9.1.2.2. Kuriranje funkcije

Kurirana funkcija je funkcija koja prima više argumenata **jedan po jedan**: umesto da primi sve argumente odjednom i vrati konačan rezultat svog izvršavanja, ona prima po jedan argument i vraća funkciju koja će da primi sledeći argument i tako redom dok se ne proslede svi argumenti. Kada se proslede svi argumenti, primena argumenata se završava i vraća se konačna vrednost. Dakle, kurirane funkcije zadovoljavaju zahtev λ-računa da funkcija prima jedan argument i vraća jedan izlaz.

U našem primeru simulacije izvršavanja funkcije $f(x,y,z)=x+y+z$ za vrednosti promenljivih $x=1$, $y=2$ i $z=3$ to bi izgledalo ovako:

1. Funkciji od tri promenljive $f(x,y,z)$ prosledi se argument $x=1$. Rezultat primene (supstitucije) argumenta je nova funkcija koja ima dve promenljive $g(y,z)$ i fiksiranu vrednost $x=1$.
2. Funkciji $g(y,z)$ prosledi se argument $y=2$. Rezultat primene ovog argumenta je nova funkcija koja ima jednu promenljivu $h(z)$ i fiksirane vrednosti $x=1$ i $y=2$.
3. Funkciji $h(z)$ prosledi se argument $z=3$. Rezultat primene ovog argumenta je vrednost $h(z)=1+2+3=6$ što je vrednost funkcije $f(x=1,y=2,z=3)=1+2+3=6$

Kada smo kroz ovu simulaciju objasnili osnovnu ideju parcijalne aplikacije i kuriranih funkcija, možemo da pređemo na primere u jeziku JavaScript kojima ćemo da ilustrujemo kako se ovo može iskoristiti.

Primer. Hoćemo da napravimo funkciju sa imenom `gte` koja treba da proveri da li je zadovoljen uslov `n>=cutoff i da vrati vrednost true` ako je uslov zadovoljen, a vrednost `false` ako uslov nije zadovoljen.

Jedna definicija funkcije korišćenjem kanoničke (posebna naredba) i streličaste[1,2] notacije mogla bi da izgleda ovako:

```
// Kanonička notacija (ključna reč function)
// gte:: a → a → b
function gte (granica, n) {
        return n >= granica;
   };

// streličasta sintaksa
// gte:: a → a → b
const gte = (granica, n) => n >= granica;
```

[1]Osnovne stvari o streličastoj notaciji možete naći na https://javascript.info/arrow-functions-basics

[2] Postoje značajne razlike između funkcija deklarisanih ključnom reči `function` i streličastom notacijom (`=>`). Funkcija deklarisana sreličastom sintaksom nema pokazivač `this`, i ne može se koristit kao konstruktor, pa o tome valja voditi računa ali je streličasta notacija veoma čitljiva

Iz ove definicije funkcije u streličastoj notaciji lako se pročita značenje funkcije `gte`: "`gte` je funkcija koja prima DVA argument `granica` i `n` tipa `number` i vraća rezultat evaluacije izraza `n >= granica`". U ovoj definiciji sama funkcija `gte` će da vrati vrednost `true` ako je `n` veće ili jednako `granica`, a vrednost `false` ako taj uslov nije zadovoljen, što znači da ona kao izlaz vraća tip `boolean`. To znači da je njena HM signatura:

```
gte:: a → a → b
```

Da bi to mogla da uradi, moraju joj se proslediti DVA argumenta.

Drugačija definicija funkcije `gte` može da izgleda ovako:

```
// Kanonička notacija (ključna reč function)
function gte (granica) {
   return function(n) {
      return n >= granica;
   };
};

// Streličasta notacija
const gte = granica => n => n >= granica;
```

Iz ove definicije funkcije u streličastoj notaciji lako se pročita značenje DRUGAČIJE funkcije `gte`: "`gte` je funkcija koja prima JEDAN argument `granica` i vraća FUNKCIJU koja prima JEDAN argument `n` i vraća rezultat evaluacije izraza `n >= granica`". U stvari, funkcija koju je vratila funkcija `gte` će da vrati vrednost `true` ako je `n` veće od ili jednako sa `granica`, a vrednost `false` ako taj uslov nije zadovoljen. Ono što je karakteristično za ovu definiciju funkcije `gte` je da ona, iako treba "na kraju dana" da obezbedi evaluiranje izraza u kome se porede DVE vrednosti kao ulaz prima JEDAN argument. Obe funkcije (sama funkcija `gte` i funkcija koju `gte` vraća su **arnosti 1 – imaju samo jedan ulazni argument)**. HM signatura je skoro ista, samo je dodat par malih zagrada na pravom mestu:

```
gte:: a → (a → b)
```

Pa da pozovemo funkciju `gte` sa vrednošću ulaza `granica = 4`:

```
const gte4 = gte(4);
```

U ovom pozivu, funkcije `gte` prosleđen je argument `granica` = 4. Rezultat poziva je **funkcija** koja prima vrednost `n`, proverava da li je zadovoljen uslov `n >= 4` i vraća vrednost `true` ako je uslov zadovoljen a vrednost `false` ako uslov nije zadovoljen.

Da bismo dobili rezultat `true` ili `false`, mora se pozvati ta vraćena funkcija i njoj se mora proslediti vrednost n. Evo kako to izgleda:

```
/* poziv funkcije gte sa ulaznom vrednošću 4 vraća funkciju gte4 koja
proverava da li je ono što joj je prosleđeno veće ili jednako 4 */
const gte4 = gte(4);

/* Sada se poziva funkcija gte4 kojoj se prosleđuje vrednost za koju se
proverava da li je veća ili jednaka 4 */
console.log(gte4(6)); // vratiće: true
console.log(gte4(3)); // vratiće: false
```

Ovo što smo upravo videli je vrlo zametno i neupotrebljivo u slučaju funkcija sa većim brojem argumenata. Rešenje je *autokuriranje* koje omogućuje automatsko kuriranje funkcija.

Sam jezik JavaScript nema ugrađen mehanizam autokuriranja (ipak JavaScript nije pravljen da bude jezik funkcionalnog programiranja) ali se on može uvesti iz različitih biblioteka, ili samostalno napraviti:

```
// Maleno, rekurzivno autokuriranje iz kućne radinosti
const curry =
```

```
(f, arr = []) => (...args) =>
(a => a.length === f.length ? f(...a) : curry(f, a))([...arr,
                                                      ...args]);
```

Za prvi primer ćemo uzeti funkciju `add3()`:

```
const add3 = curry((a, b, c) => a + b + c);
```

Funkcija koja se ovde kurira je funkcija `(a, b, c) => a + b + c`

Pomoću autokuriranja funkcija `add3` se može koristiti na više različitih načina, zavisno od argumenata koji joj se proslede:

```
console.log (add3(1, 2, 3)); // 6
console.log (add3(1, 2)(3)); // 6
console.log (add3(1)(2, 3)); // 6
console.log (add3(1)(2)(3)); // 6
```

Da vidimo sada kako će nam "*Maleno, rekurzivno autokuriranje*" pomoći da dva argumenta u prvoj definiciji funkcije `gte` obrađujemo jedan po jedan:

```
const gte2 = curry((granica, n) => n >= granica);

console.log(gte2((6),(4))); // vratiće: false
console.log(gte2((3),(4))); // vratiće: true
console.log(gte2((33),(44))); // vratiće: true
console.log(gte2((33),(22))); // vratiće: false
```

Na ovaj način dobili smo funkciju `gte2` koja je kurirana verzija funkcije `gte` – prosleđuju joj se dva argumenta, a ona ih obrađuje jedan po jedan.

9.1.3. Čiste funkcije

Pitanje "čistote" je jedno od ključnih pitanja funkcionalnog programiranja. Kako kaže Majkl Fogus u [JavaScript functional programming]: "Funkcionalno programiranje nije samo funkcija; to je i način razmišljanja kako izgraditi programe da bi se minimizirala složenost koja je svojstvena kreiranju softvera. Jedan od načina da se smanji složenost je smanjenje ili (idealno) eliminisanje otiska promene stanja koja se dešava u našim programima." Taj otisak promene stanja naziva se bočnim efektom.

Filozofija funkcionalnog programiranja postavlja kao polazište stav da su bočni efekti primarni uzrok nekorektnog ponašanja programa. Pošto se ponašanje u funkcionalnom programiranju implementira putem funkcija, jasno je da u tome, ipak, najznačajniju ulogu ima funkcija. U ovom odeljku objasnićemo šta je čista funkcija i na koji način se ona može ostvariti u JavaScript-u.

Počećemo sa definicijom čiste funkcije: ***Čista funkcija je funkcija koja za zadati ulaz uvek vraća isti izlaz i nema vidljivih bočnih efekata***.

Lepo, imamo dobru definiciju ali je pitanje kako napraviti funkciju koja je zadovoljava? Da bismo to uradili, treba da znamo šta tačno znače termini "***za zadati ulaz uvek vratiti isti izlaz***" i "***bočni efekat***".

Prvo ćemo da preciziramo značenje fraze "***za zadati ulaz uvek vratiti isti izlaz***".

Odgovor na ovo pitanje je u značenju funkcije u funkcionalnom programiranju koje je ekvivalentno definiciji funkcije u matematici a koja kaže: **Funkcija je specijalna relacija između vrednosti koja SVAKOJ ULAZNOJ VREDNOSTI DODELJUJE TAČNO JEDNU IZLAZNU VREDNOST.**

Dakle, i funkcija u JavaScript-u treba da za svaku datu kombinaciju ulaznih argumenata vrati ISTI IZLAZ. Ovim dolazimo do jednog principa funkcionalnog programiranja koji se zove ***referencijalna transparentnost*** i koji glasi: referencijalno transparentna je ona funkcija koja za zadati ulaz uvek vraća isti (odgovarajući) izlaz. Referencijalna transparentnost omogućuje da se funkcija na svakom mestu u

programu može zameniti svojim izlazom a da ponašanje programa ostane isto. Ona je osnova mnogih važnih mogućnosti kao što je, na primer, memoizacija[3].

Sada ćemo da preciziramo značenje termina "***bočni efekat***".

Definicija bočnog efekta je: ***Bočni efekat je promena stanja sistema (sadržaja memorije programa koji se izvršava) ili vidljiva interakcija sa spoljašnjim svetom koja se dešava u toku sračunavanja rezultata***.

Primeri bočnih efekata su:

- Promene fajl sistema (kreiranje, brisanje, modifikovanje sadržaja fajlova);
- Umetanje novog zapisa u bazu podataka;
- Http poziv;
- Ispisi na ekran/logovanja;
- Prihvatanje ulaza od korisnika;
- Pravljenje upita nad DOM stablom;
- Pristup stanju sistema;
- ……..

U stvari, svaka interakcija funkcije sa spoljašnjim svetom je bočni efekat pa se postavlja pitanje: Šta, uopšte, može korisno da uradi program koji NEMA BOČNIH EFEKATA? Odgovor je: NIŠTA.

A pošto nama trebaju programi koji rade korisne stvari, ideja funkcionalnog programiranja NIJE DA SE POTPUNO UKINU BOČNI EFEKTI nego DA SE BOČNI EFEKTI KONTROLIŠU.

ČISTE FUNKCIJE su mehanizam za KONTROLISANJE BOČNIH EFEKATA.

Sada možemo da kažemo kaže koje konkretne osobine treba da ima čista funkcija pa da za zadati ulaz uvek vrati isti izlaz i da ne proizvede vidljive bočne efekte.

Odgovor je: Čista funkcija je ona funkcija koja zadovoljava sledeće uslove:

- Njen rezultat (izlazna vrednost) sračunava se samo na osnovu vrednosti njenih argumenata.
- Ona ne može da se oslanja na podatke koji se menjaju "spolja", izvan njene kontrole.
- Ona ne može da menja stanje nečega što je izvan njenog tela – eksternog stanja.

Da bismo bolje razumeli značenje termina bočni efekat, sada ćemo da pokažemo primere funkcija pisanih u JavaScript-u koje su "nečiste" i primere istih tih funkcija transformisanih u "čiste" funkcije.

Primer 1: Bez eksplicirane strategije

U ovom primeru pokazaćemo kod koji kao bočni efekat proizvodi neželjenu promenu eksternog stanja, dok su prva dva uslova koje treba da zadovoljava čista funkcija ispunjena. Funkcija `dodajUkorpu()` je ovde nečista funkcija.

```
/* Na početku objekat originalKorpa ima jedno svojstvo stavke koje je prazan niz. */

const originalKorpa = {
      stavke: []
};
IspisiNaslov("U ovom primeru funkcija dodajUkorpu je nečista ")
IspisiNaslov("originalKorpa pre poziva funkcije dodajUkorpu")
IspisiObjekat(originalKorpa);
```

[3] Memoizacija je način predstavljanja funkcije putem tabele u kojoj argumenti formiraju ključeve koji pokazuju na sračunatu vrednost. Time se omogućuje smanjenje broja izračunavanja funkcije. Umesto da se funkcija izvršava svaki put, njena vrednost se uzima iz tabele ako je već izračunata za datu konfiguraciju argumenata.

```
/* Nečista funkcija dodajUkorpu napisana je tako da mutira postojeći objekat
korpa, preciznije mutira njegovo svojstvo stavke */
const dodajUkorpu = (korpa, stavka, kolicina) => {
      korpa.stavke.push({
            stavka,
            kolicina
      });
            return korpa;
};

// Prvi poziv funkcije dodajuKorpu
let novaKorpa = dodajUkorpu(
      originalKorpa,
{
      naziv: "Digitalna SLR kamera",
      cena: '1495'
},
1
);
IspisiNaslov("novaKorpa nakon prvog poziva funkcije dodajUkorpu ")
IspisiObjekat(novaKorpa);
IspisiNaslov("originalKorpa nakon prvog poziva funkcije dodajUkorpu ")
IspisiObjekat(originalKorpa);

// Drugi poziv funkcije dodajuKorpu
novaKorpa = dodajUkorpu(
      originalKorpa,
{
      naziv: "Muška košulja",
      cena: '12.3'
},
2
);
IspisiNaslov("novaKorpa nakon drugog poziva funkcije dodajUkorpu ")
IspisiObjekat(novaKorpa);
IspisiNaslov("originalKorpa nakon drugog poziva funkcije dodajUkorpu ")
IspisiObjekat(originalKorpa);
```

Log iz ovog programa je:

U ovom primeru funkcija dodajUkorpu je nečista
originalKorpa pre poziva funkcije dodajUkorpu

```
{
 "stavke": [ ]
}
```

novaKorpa nakon prvog poziva funkcije dodajUkorpu

```
{
 "stavke": [
  {
   "stavka": {
    "naziv": "Digitalna SLR kamera",
    "cena": "1495"
   },
```

```
    "kolicina": 1
   }
  ]
}
```


originalKorpa nakon prvog poziva funkcije dodajUkorpu


```
{
  "stavke": [
   {
    "stavka": {
     "naziv": "Digitalna SLR kamera",
     "cena": "1495"
    },
    "kolicina": 1
   }
  ]
}
```


novaKorpa nakon drugog poziva funkcije dodajUkorpu


```
{
  "stavke": [
   {
    "stavka": {
     "naziv": "Digitalna SLR kamera",
     "cena": "1495"
    },
    "kolicina": 1
   },
   {
    "stavka": {
     "naziv": "Muška košulja",
     "cena": "12.3"
    },
    "kolicina": 2
   }
  ]
}
```


originalKorpa nakon drugog poziva funkcije dodajUkorpu


```
{
  "stavke": [
   {
    "stavka": {
     "naziv": "Digitalna SLR kamera",
     "cena": "1495"
    },
    "kolicina": 1
   },
   {
    "stavka": {
     "naziv": "Muška košulja",
     "cena": "12.3"
    },
    "kolicina": 2
```

```
  }
 ]
}
```

Preradićemo našu nečistu funkciju `dodajUkorpu()` u novu funkcija `dodajUkorpu()` koja ne mutira postojeći objekta **korpa**. Nova funkcija je:

```
/* Čista funkcija dodajUkorpu koja ne mutira postojeći objekat korpa, preciznije ne mutira njegovo svojstvo stavke */
const dodajUkorpu = (korpa, stavka, kolicina) => {
      return {
            ...korpa,
            stavke: korpa.stavke.concat([{
              stavka,
              kolicina
            }]),
      };
      };
```

Ako u prethodnom kodu zamenimo postojeći kod funkcije dodajUKorpu novim kodom i liniju `IspisiNaslov("U ovom primeru funkcija dodajUkorpu je nečista ")` zamenimo linijom `IspisiNaslov("U ovom primeru funkcija dodajUkorpu je čista ")`, dobićemo sledeći log:

```
U ovom primeru funkcija dodajUkorpu je čista
originalKorpa pre poziva funkcije dodajUkorpu
{
 "stavke": [ ]
}
novaKorpa nakon prvog poziva funkcije dodajUkorpu
{
 "stavke": [
  {
   "stavka": {
    "naziv": "Digitalna SLR kamera",
    "cena": "1495"
   },
   "kolicina": 1
  }
 ]
}
originalKorpa nakon prvog poziva funkcije dodajUkorpu
{
 "stavke": [ ]
}
novaKorpa nakon drugog poziva funkcije dodajUkorpu
{
 "stavke": [
  {
   "stavka": {
    "naziv": "Digitalna SLR kamera",
    "cena": "1495"
   },
```

```
    "kolicina": 1
   },
   {
    "stavka": {
     "naziv": "Muška košulja",
     "cena": "12.3"
    },
    "kolicina": 2
   }
 ]
}
```


originalKorpa nakon drugog poziva funkcije dodajUkorpu


```
{
 "stavke": [ ]
}
```


U čemu je razlika između dve verzije funkcija `dodajUkorpu()`?

I jedna i druga imaju tri ULAZNA parametra: `korpa`, `stavka` i `kolicina` i tu nema nikakve razlike među njima.

Međutim, logike formiranja objekta `korpa` su im različite.

Prva verzija funkcije dodajUKorpu dodaje vrednosti `stavka` i `kolicina` **direktno u niz `stavke`** koji je svojstvo objekta `korpa` - ugrađena metoda `push()` dodaje specificirani element na kraj ulaznog niza modifikujući na taj način ulaz (ovde je to niz **`stavke`** koji je svojstvo objekta **`korpa`**).:

```
korpa.stavke.push({
            stavka,
            kolicina
      })
```

Druga verzija funkcije `dodajUkorpu()` ne dodaje vrednosti `stavka` i `kolicina` direktno u svojstvo **`stavke`** objekta **`korpa`** nego kreira novi niz **`stavke`** - ugrađena metoda `concat()` vrši spajanje nizova tako što kreira novi niz a ne modifikuje polazne nizove:

```
      return {
            ...korpa,
            stavke: korpa.stavke.concat([{
              stavka,
              kolicina
            }]),
      };
```

Primer 2: Strategija injekcije zavisnosti

U softverskom inženjerstvu, injekcija zavisnosti je tehnika programiranja u kojoj objekat ili funkcija prima druge objekte ili funkcije koje su mu potrebne, umesto da ih interno kreira.

U ovom primeru ilustrovaćemo situaciju u kojoj je nečistoća funkcije uzrokovana činjenicom da funkcija radi skoro sve što čistoj funkciji nije dozvoljeno: svoj izlaz kreira na bazi ulaza koji joj nisu prosleđeni putem argumenata i na bazi ponašanja koje je izvan nje same te ostvaruje interakciju sa spoljašnjim okruženjem . Primer je jednostavan: funkcija koja na konzoli ispisuje log koji sadrži nekakav tekst i datum i vreme u ISO formatu.

Prva, nečista, varijanta ove funkcije data je kodom:

```
// Nečista funkcija
// logujNesto :: String -> String
```

```
function logujNesto(nesto) {
    const dt = new Date().toISOString();
    console.log(`${nesto}: ${dt} `);
    return nesto;
}
logujNesto(' Ispisujem ovu poruku u trenutku')
```

Signatura kaže da je to funkcija koja prima string i vraća string. Ako pogledate kod, vidite da funkcija vraća string koji je primila. I tu nema ničega nečistog – vraća ono što je primila putem svog argumenta.

Međutim, funkcija pre vraćanja izlaza radi stvari koje u nju unose nečistoću.

Prvo što uradi je da kreira novi objekat i to tako što pozove konstruktor objekta koji joj nije prosleđen kao argument i pozove njegovu metodu (konstruktor `new Date()`i metoda `toISOString()`). To znači, oslanja se na ponašanja izvan nje same. Takođe, ona izvršava i metodu `log()` objekta `console` koji joj takođe nije prosleđen putem argumenta.

A evo kako izgleda (jedna) čista varijanta funkcije `logujNesto()`:

```
// Čista funkcija
// logujNesto: String -> Date -> Console -> *

function logujNesto(nesto, d, cnsl) {
    const dt = d.toISOString();
    return cnsl.log(`${nesto}: ${dt}}`);
}
```

U novoj verziji funkcija svoj rezultat kreira samo na osnovu onoga što je primila kroz argumente. Umesto jednog parametra, ona sada ima tri parametra. Prvi je string (`nesto`) koji je kao parametar imala i nečista verzija funkcije, drugi je objekat datumskog tipa, a treći je objekat koji treba da ima pristup metodi `log()`. Ona ne vraća nikakvu posebnu vrednost, prosto uradi ono što piše u njenom telu i vrati kontrolu izvršavanja onome ko ju je pozvao. To se sve jasno vidi kada se izvrši sledeći snipet:

```
const datumVreme = new Date();
let Poruka = 'Ispisujem ovu poruku u trenutku'
logujNesto( datumVreme, console, Poruka)
let vraceno = logujNesto( datumVreme, console, Poruka)
console.log('vracena vrednost funkcije je: ', vraceno) // vratiće undefined
```

Ovakav pristup kojim se nečista funkcija `logujNesto()` učini čistom zove se ***injekcija zavisnosti***.

Injekcija zavisnosti je, dakle, pristup u kome se uzimaju sve nečistoće u kodu i „guraju“ u parametre funkcije. Na taj način, nečistoće postaju „tuđa odgovornost“ – odgovornost nekih drugih funkcija. Dosta uobičajeno ponašanje, ne samo u funkcionalnom programiranju nego i u svakodnevnom životu. Naravno, nečistoća na kraju dođe na naplatu.

Međutim, injekcija zavisnosti obezbeđuje jednu važnu stvar: ako hoćete da uradite nešto nečisto, morate to da učinite nečistim.

Evo i primera: Ako našoj funkciji `logujNesto()` zadamo ovakve parametre:

```
const d = {toISOString: () => '1865-11-26T16:00:00.000Z'};
const cnsl = {
    log: () => {
        // nemoj ništa da radiš
    },
};
```

njen poziv

```
logujNesto(d, cnsl, "Pozdravi ga puno!");
```

neće uraditi ništa. I svaki put kada je pozovemo sa ovim ulazima isto će da se desi – ona neće uraditi ništa.

Glavni nedostatak injekcije zavisnosti je što ona dovodi do enormno dugačkih signatura funkcija što može da bude veoma zamorno.

Primer 3: Strategija odlaganja

Pored injekcije zavisnosti, za upravljanje bočnim efektima u praksi se koristi i pristup zvani ***odlaganje***. Ovaj pristup zasniva se na postulatu: ***bočni efekat nije bočni efekat dok se ne desi***. Dok se ne desi, on neće napraviti nikakav efekat (ni loš ni dobar).

U osnovi, ovaj pristup koristi indirekciju pomoću omotačkih funkcija kojom se odlaže evaluacija funkcije. Iako smo već rekli da indirekcija pomoću omotačkih funkcija nije baš srećno rešenje za one koji će kod čitati, održavati i, eventualno, dalje razvijati, postoji u JavaScript-u mehanizam funktora koji omogućuje da se to odlaganje uradi na elegantan način koji rezultuje vrlo čitljivim kodom podesnim za održavanje. O funktorima ćemo posebno govoriti kasnije i tom prilikom ćemo da pokažemo i kako se oni mogu koristiti za upravljanje bočnim efektima.

Sada ćemo da pokažemo implementaciju tehnike odlaganja koju možemo da razumemo sa onim što do sada znamo o funkcijama. Primer je preuzet iz [Sinclair] i malo je prilagođen da se naglasi osnovna ideja odlaganja. Implementirana je "odložena" aritmetika u kojoj se operacije izvršavaju tek kada se pozovu funkcije koje "pozivaju" numeričke vrednosti. Sledi kod:

```
// "Odložena" aritmetika
// vratiNulaFunk :: () -> (() -> Number)
function vratiNulaFunk() {
    function fNula() {
        return 0;
    }
    return fNula;
}

// fIncrement :: (() -> Number) -> (() -> Number)
function fIncrement(f) {
    return () => f() + 1;
}

// Aritmetika
// fMnozi :: (() -> Number) -> (() -> Number) -> (() -> Number)
function fMnozi(a, b) {
         return () => a() * b();
}

// fStepen :: (() -> Number) -> (() -> Number) -> (() -> Number)
function fStepen(a, b) {
       return () => Math.pow(a(), b());
      }

// fKoren :: (() -> Number) -> (() -> Number)
function fKoren(x) {
       return () => Math.sqrt(x());
      }

// "Odložena" računanje
```

```
// Ovo što sledi su "odloženi" brojevi
const fNula  = vratiNulaFunk(); // vraća funkciju koja će vratiti 0 kada se
izvrši
const fJedan = fIncrement(fNula); // vraća funkciju koja će vratiti 1 kada
se izvrši
const fDva   = fIncrement(fJedan) // vraća funkciju koja će vratiti 2 kada
se izvrši
const fTri   = fIncrement(fDva) // vraća funkciju koja će vratiti 3 kada se
izvrši
const fCetiri= fIncrement(fTri) // vraća funkciju koja će vratiti 4 kada se
izvrši

console.log ('\"Odloženi\" brojevi \n ')
console.log('\"Odloženo\" 0: ', fNula);
console.log('\"Odloženo\" 1: ', fJedan);
console.log('\"Odloženo\" 2: ', fDva);
console.log('\"Odloženo\" 3: ', fTri);
console.log('\"Odloženo\" 4: ', fCetiri);
console.log ('\n')

// Sada "odloženo" računamo
console.log ('Ovo su \"odložena\" računanja - omotačke funkcije  ')

const DvaNaKvadrat = fStepen(fDva,fDva) // omotačka funkcija za 2**2
console.log('Dva na kvadrat: ', DvaNaKvadrat)
const CetiriPutaDva = fMnozi(fCetiri,fDva); // omotačka funkcija za 4*2
console.log('Četiri puta dva: ', CetiriPutaDva)
const TriNaTri = fStepen(fTri, fTri); // omotačka funkcija za 3**3
console.log('Tri na treći: ', TriNaTri)
const KorenIz27 = fKoren(TriNaTri); // omotačka funkcija za koren iz 27
console.log('Koren iz dvadesetsedam: ', KorenIz27)
console.log ('\n')

// A tek sada se stvarno računa - omotačke funkcije se pozivaju
console.log ('Ovo su rezultati računanja - pozivanja omotačkih funkcija nad
\"odloženim\"  brojevima ')

console.log('Dva na kvadrat: ', DvaNaKvadrat()) // poziva funkciju i vraća
4
console.log('Četiri puta dva: ', CetiriPutaDva())// poziva funkciju i vraća
8
console.log('Tri na treći: ', TriNaTri())// poziva funkciju i vraća 27
console.log('Koren iz dvadesetsedam: ', KorenIz27()) // poziva funkciju i
vraća 5.196152422706632
```

Log izgleda ovako:

"Odloženi" brojevi

"Odloženo" 0: ƒ fNula() {
**   return 0;**
**   }**
"Odloženo" 1: () => f() + 1
"Odloženo" 2: () => f() + 1

"Odloženo" 3: `() => f() + 1`
"Odloženo" 4: `() => f() + 1`

Ovo su "odložena" računanja - omotačke funkcije
Dva na kvadrat: `() => Math.pow(a(), b())`
Četiri puta dva: `() => a() * b()`
Tri na treći: `() => Math.pow(a(), b())`
Koren iz dvadesetsedam: `() => Math.sqrt(x())`

Ovo su rezultati računanja - pozivanja omotačkih funkcija nad "odloženim" brojevima
Dva na kvadrat: 4
Četiri puta dva: 8
Tri na treći: 27
Koren iz dvadesetsedam: 5.196152422706632

9.1.4. Funkcije višeg reda i kompozicija funkcija

Kompozicija funkcija je osnovni mehanizam apstrakcije ponašanja u funkcionalnom programiranju. Osnovni mehanizam za kompoziciju funkcija su funkcije višeg reda.

Programski jezik JavaScript podržava nativno funkcije višeg reda i time omogućuje implementaciju različitih postupaka kompozicije funkcija.

U ovom odeljku govorićemo o funkcijama višeg reda i kompoziciji funkcija sa naglaskom na funkcije višeg reda i kompoziciju funkcija u JavaScript-u.

9.1.4.1. Funkcije višeg reda u matematici i računarstvu

U matematici i računarskoj nauci, ***funkcija višeg reda*** (higher-order function, HOF) je funkcija koja poseduje bar jedno od sledećih svojstava:

- Može da primi jednu ili više funkcija kao argumente.
- Može da vrati funkciju kao rezultat.

Sve druge funkcije su ***funkcije prvog reda***.

Vrlo očigledan primer funkcije višeg reda u matematici je operator diferenciranja (izvod). On prima funkciju kao ulaz i vraća kao izlaz drugu funkciju koja je izvod ulazne funkcije. Na primer:

$$diff(f = x^2 + 7x + 1) = g(x) = 2x + 7$$

U netipiziranom **λ**-računu sve funkcije su funkcije višeg reda. U tipiziranom **λ**-računu iz koga je izvedena većina funkcionalnih programskih jezika, funkcije višeg reda koje primaju neku funkciju kao argument su vrednosti sa tipovima oblika:

$$(\tau_1 \to \tau_2) \to \tau_3$$

Funkcije višeg reda podržane su u gotovo svim programskim jezicima, pa i u JavaScript-u. Iako smo do sada u primerima prećutno obilno koristili funkcije višeg reda, evo i eksplicitnih primera koji ilustruju funkcije višeg reda u JavaScript-u u kanoničkoj i streličastoj sintaksi.

```
// Streličasta sintaksa
"use strict";

const dvaPuta = f => x => f(f(x));
const dodajTri = i => i + 3;
const g = dvaPuta(dodajTri);
```

```
console.log(g(15)); // 21

// Kanonička sintaksa
"use strict";

function dvaPuta (f) {
  return function (x) {
    return f(f(x));
  };
}

function dodajTri (i) {
  return i + 3;
}

const g = dvaPuta (dodajTri);
console.log(g(20)); // 26
```

9.1.4.1.1. Funkcije map(), filter() i reduce()

Vrlo važne funkcije višeg reda u JavaScript-u su funkcije `map()`, `filter()` i `reduce()`. Ovde ćemo ih samo ukratko objasniti a kasnije će o njima biti još reči.

Funkcija `map()` je funkcija koja prima dva argumenta od kojih je jedan funkcija a drugi niz i vraća kao izlaz niz. Dakle, HM signatura funkcije `map()` mogla bi da izgleda ovako:

```
map:: (a → [b]) → [b]
```

Funkcija primenjuje ulaznu funkciju na ulazni niz i kao rezultat vraća novi izlazni niz koji sadrži transformisane elemente ulaznog niza kao u sledećem primeru (ovde je pretpostavka da je funkcija `map()` prethodno definisana):

```
const double = x => x * 2;
const ulazniNiz = [1, 2, 3];
let izlazniNiz = map(double, ulazniNiz ) // [2, 4, 6]
```

Funkcija `filter()` je funkcija koja prima dva argumenta od kojih je jedan funkcija a drugi niz i vraća kao izlaz niz. Dakle, HM signatura funkcije `filter()` mogla bi da izgleda ovako:

```
filter:: (b → [a]) → [b]
```

Ulazna funkcija (zove se i predikat) je funkcija koja vraća `true` ili `false`. Oni članovi ulaznog niza koji zadovoljavaju uslov iskazan predikatskom funkcijom (funkcija vrati `true`) prepisuju se u izlazni niza po redosledu pristizanja. Evo primera (i ovde je pretpostavka da je funkcija `filter()` prethodno definisana):

```
const VeceOdDva = x => x > 2;
const ulazniNiz = [1, 2, 3, 4, 5];
let izlazniNiz = filter(VeceOdDva, ulazniNiz ) // [3, 4, 5]
```

JavaScript ima ugrađene funkcije `map()` i `filter()` za nizove. Detalji se mogu naći na https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map. Koristićemo ih u primerima koji slede u kojima nam je cilj da ilustrujemo mogućnosti funkcija višeg reda.

Primer [Eliot]: U ovom primeru prikazana je funkcija **prvog reda** koja formira novi niz u koji smešta sve reči iz polaznog niza koje imaju broj karaktera različit od 4.. Evo koda:

```
const ManjeViseOd4Znaka = 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;
};

let mojeReci = ['oops', 'radnik', 'rame', 'automobil']
let mojeReciNije4Znaka = ManjeViseOd4Znaka (mojeReci);
console.log('U nizu reči: ', mojeReci,'\n reči koje nisu od 4 slova su: \n',
mojeReciNije4Znaka)
```

Druga funkcija prvog reda iz zadatog niza u novi niz smešta sve reči koje počinju slovom 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;
};

let mojeReci = ['Sat', 'spasenje', 'jedrilica', 'svetionik']
let mojeRecipocinjuSaS = pocinjeSaS (mojeReci);
console.log('U nizu reči: ', mojeReci,'\n reči koje počinju sa \"s" su: \n',
mojeRecipocinjuSaS)
```

Ono što je uočljivo u kodu funkcija u ova dva programa je da se mnogo stvari ponavlja. Ako izuzmemo imena funkcija, one se razlikuju samo u po jednoj liniji koda (linije 5 u oba koda). Globalno gledano, obe ove funkcije rade istu stvar: Iteriraju nad nizom i iz njega izdvajaju elemente koji zadovoljavaju zadati uslov. Jedino što je u njima različito je taj uslov. A uslov je funkcija koja vraća logičku vrednost true ili false. To znači da bi preuređena funkcija koja taj uslov prima kao argument mogla da radi oba ova posla. A preuređena funkcija bi mogla da izgleda ovako:

```
const izdvojiPotrebno = (uslov,lista) => {
      const filtrirano = [];
      for (let i = 0, { length } = lista; i < length; i++) {
            const element = lista[i];
            if (uslov(element)){ filtrirano.push(element)};
      }
                  return filtrirano;
};
```

Sada dva zadatka možemo rešiti tako što ćemo definisati funkcije koje opisuju odgovarajuće uslove:

```
// Uslov za izdvajanje reči koje imaju broj znakova različit od 4
const uslov1 = ulaz => ulaz.length !== 4

// Uslov za izdvajanje reči koje počinju znakom s
const uslov2 = ulaz => ulaz.startsWith('s')
```

Nakon toga poziva se funkcija `izdvojiPotrebno()` kojoj se prosleđuje odgovarajući uslov:

```
// Poziv za izdvajanje reči koje imaju broj znakova različit od 4
```

```
let mojeReci1 = ['oops', 'radnik', 'rame', 'automobil']
let mojeReciNije4Znaka = izdvojiPotrebno (uslov1, mojeReci1);
console.log('U nizu reči: ', mojeReci,'\n reči koje nisu od 4 slova su: \n',
mojeReciNije4Znaka)

// Poziv za izdvajanje reči koje počinju znakom s
let mojeReci2 = ['Sat', 'spasenje', 'jedrilica', 'svetionik']
let mojeRecipocinjuSaS = izdvojiPotrebno (uslov2, mojeReci2);
console.log('U nizu reči: ', mojeReci,'\n reči koje počinju sa \"s" su: \n',
mojeRecipocinjuSaS)
```

Ovo što smo uradili u primeru je primena funkcija višeg reda kojom smo napravili prilično naivnu generalizaciju zadatka i njegovog rešenja sa dosta zaostalih specifičnosti koje ograničavaju ponovnu upotrebu naše funkcije `izdvojiPotrebno()`. Sa još malo koncentracije i truda dobija se funkcija sa znatno širim mogućnostima korišćenja. To je funkcija **`reduce().`**

Funkcija `reduce()` je funkcija koja prima dva obavezna argumenta od kojih je jedan funkcija a drugi niz i jedan opcioni argument koji predstavlja početnu vrednost. Kao izlaz vraća jednu vrednost. Dakle, HM signatura funkcije `filter()` mogla bi da izgleda ovako:

```
// reduce :: (b -> a -> b) -> b -> [a] -> b
```

Funkcija `reduce()` "akumulira" vrednosti iz ulaznog niza tako što kaskadno primenjuje primljenu funkciju na elemente niza. Ukoliko je primila opcioni argument, u prvom koraku akumulira tu vrednost sa prvim članom niza, a ukoliko nije primila opcioni argument počinje akumuliranje sa prvim članom niza.

```
const reduce = (reducer, initial, arr) => {
// Zajednički deo
let acc = initial;
      for (let i = 0, { length } = arr; i < length; i++) {

// deo koji je jedinstven (specifičan) za poziv funkcije reducer
            acc = reducer(acc, arr[i]);

// Još malo zajedničkog
      }
            return acc;
};
```

Ovu funkciju možemo iskoristiti da, na primer, saberemo članove niza prosleđujući odgovarajuću funkciju `reducer` (u sledećem pozivu to je funkcija `(acc, curr) => acc + curr`) i početnu vrednost `initial` (u pozivu je to 0):

```
reduce((acc, curr) => acc + curr, 0, [10,20,30]); // 60
```

Korišćenjem funkcije `reduce()` možemo da napravimo i `filter()`:

```
const filter = (
      fn, arr
      ) => reduce((acc, curr) => fn(curr) ?
      acc.concat([curr]) :
      acc, [], arr
);
const fn = ulaz => ulaz > 1
filter(fn,[1,2,3]); // [2,3]
```

Sada kada imamo filter možemo jednostavno da napravimo kod za naše “igranje sa rečima”:

```
const ManjeViseOd4Znaka = reci => filter(
rec => rec.length !== 4,
reci
);
ManjeViseOd4Znaka(['oops', 'radnik', 'rame', 'automobil']) /* ['radnik',
'automobil'] */

const PocinjeSaS = reci => filter(
rec => rec.startsWith('s'),
reci
);
PocinjeSaS(['Sat', 'spasenje', 'jedrilica', 'svetionik']) /* ['spasenje',
'svetionik'] /*
```

Funkcije višeg reda imaju još jednu važnu primenu: apstrahovanje načina rada nad različitim tipovima podataka. Na primer, ista funkcija `filter()` ne mora da radi samo sa nizovima stringova, nego može lako da filtrira i brojeve kao u sledećem primeru:

```
const VeciOd = cutoff => n => n >= cutoff;
const VeciOd3 = VeciOd(3);
[1, 2, 3, 4].filter(VeciOd3); // [3, 4];
```

Drugim rečima, funkcije višeg reda mogu se koristiti da se funkcija učini polimorfičnom – pri svakom pozivu funkciji se može proslediti željeno ponašanje. Funkcije višeg reda su nativno polimorfične.

Iz svega ovoga sledi da su funkcije višeg reda mnogo upotrebljivije i svestranije od svojih „rođaka“ prvog reda.

9.1.4.2. Kompozicija funkcija

Kompozicija funkcija je srž funkcionalnog programiranja.

9.1.4.2.1. Kuriranje i kompozicija funkcija

Prethodno smo u odeljku Parcijalna aplikacija i kuriranje funkcija izložili ideje parcijalne aplikacije i kuriranja. Da se podsetimo.

Parcijalna aplikacija funkcije je aplikacija u kojoj se funkciji prosleđuje vrednosti (argumenti) samo dela njenih parametara a ne vrednosti svih parametara. Kao rezultat parcijalna aplikacija vraća funkciju koja zavisi od preostalih (neprosleđenih) parametara.

Kurirana funkcija je funkcija koja **svoje argumente prima jedan po jedan** vraćajući kao rezultat **unarnu funkciju** sve dok se ne prosledi poslednja vrednost parametra, odnosno dok se ne evaluira poslednja vraćena funkcija. Rezultat te poslednje evaluacije je izlaz kurirane funkcije.

Unarnost je **suštinska odlika kurirane funkcije** – svaka vraćena funkcija ima samo jedan parametar, odnosno prima samo jedan argument.

Iz ovoga sledi da je **kuriranje specijalan slučaj parcijalne aplikacije -** parcijalna aplikacija koja **uvek vraća unarnu funkciju**. **Parcijalna aplikacija u opštem slučaju nije kuriranje** jer parcijalna aplikacija **nije unarna** funkcija – ona može da vrati funkciju koja prihvata više argumenata odjednom.

9.1.4.2.1.1. Point-free stil

Svi načini definisanja funkcija koje smo do sada prikazali zahtevaju da se u definiciji funkcije navedu njeni parametri:

```
function foo (/* Ovde se deklarišu parametri */) {
```

```
// telo funkcije
}

const foo = (/* Ovde se deklarišu parametri */) => // telo funkcije

const foo = function (/* Ovde se deklarišu parametri */) {
// telo funkcije
}
```

Point-free je stil programiranja u kome se **u definicijama funkcija ne navode parametri funkcije**.

Kako je, uopšte, moguće definisati funkciju bez navođenja parametara a da joj, ipak, možemo proslediti ulaze? Mehanizam koji nam to omogućuje je funkcija višeg reda, odnosno parcijalna aplikacija - da se kreira funkcija koja vraća funkciju koja će prihvatiti vrednosti parametara.

Evo primera.

Definisaćemo funkciju `saberi()` na sledeći način:

```
// saberi:: a → (b → a + b)
const saberi = a => b => a + b;
```

Naša funkcija `saberi()` ovako definisana je, u stvari, parcijalna aplikacija (sasvim precizno, kurirana funkcija). Ona prima kao ulaz vrednost prvog sabirka i vraća funkciju sa fiksiranom vrednošću prvog sabirka. Vraćena funkcija prima kao ulaz vrednost drugog sabirka i izračunava zbir. Možemo je pozvati na sledeći način:

```
saberi(2)(3); // 5
```

Ovde poziv `saberi(2)` vraća funkciju koja izgleda ovako:

```
parcSaberi = b => 2 + b
```

A drugi poziv je, u stvari, `parcSaberi(3)` koji vraća 2+3 = 5.

Sada tu funkciju možemo iskoristiti da definišemo drugu funkciju bez navođenja parametara. Na primer, možemo definisati funkciju `inc` bez parametara koja će povećati svaku prosleđenu vrednost za 1:

```
// inc = n => Number
// Dodaje 1 na bilo koji broj.
const inc = saberi(1); // funkcija inc nema parametara
inc(7); // ovaj poziv vratiće ipak rezultat 8
```

Dakle, funkciju `inc` deklarisali smo bez parametara a u pozivu smo joj prosledili vrednost 7 kao argument i ona nam je vratila 8 što je i očekivano.

Kako se to desilo?

Pa desilo se tako što smo mi našu funkciju `inc` definisali kao **parcijalnu aplikaciju funkcije `saberi()`** u kojoj smo fiksirali prvi parametar na 1. Sada se poziva ta funkciji koju je vratila parcijalna aplikacija funkcije `saberi()` pozivom `inc(7)` i ona vrednost 7 dodaje na 1.

Ovo je zgodan mehanizam za generalizaciju i specijalizaciju. Vraćena funkcija je, u stvari, specijalizovana verzija opštije funkcije `saberi()`. Tu opštiju funkciju možemo koristiti da kreiramo više različitih specijalizovanih funkcija. Na primer, ovako:

```
const inc10 = saberi(10);
const inc20 = saberi(20);
inc10(3); // => 13
inc20(3); // => 23
inc(7); // => 8
```

Sve ove funkcije imaju sopstvena zatvaranja (zatvaranja se kreiraju pri kreiranju funkcije kada se poziva funkcija `saberi()`) pa zbog toga originalna `inc()` funkcija nastavlja da radi.

9.1.4.2.1.2. Zašto se vrši kuriranje

Razlog kuriranja leži u činjenici da su kurirane funkcije posebno korisne u kompoziciji funkcija. Pa da objasnimo ovo tvrđenje.

Posmatrajmo kompoziciju funkcija u matematici. Neka su nam zadate dve funkcije $f(x)$ i $g(x)$. Kompozicija ($\circ$) funkcija f i g je funkcija h definisana na sledeći način:

$$h(x) = f \circ g(x) = f\big(g(x)\big).$$

Isto je i u JavaScript-u:

```
const g = n => n + 1;
const f = n => n * 2;
console.log(f(g(20)));          //=> 42

const h = x => f(g(x));
console.log(h(20));             //=> 42
```

Na ovaj način komponovali smo dve funkcije.

U opštem slučaju matematika nam omogućuje da komponujemo onoliko funkcija koliko želimo. Neka su date funkcije $f_1(x)$, $f_2(x)$, . . ., $f_n(x)$. Njihova kompozicija je funkcija

$$h(x) = f_1\left(f_2\big(\,.\,.\,.f_n(x)\big)\right),$$

Isto to nam je potrebno i u JavaScript-u. Kako ga možemo napraviti? Naravno, kuriranjem u kome se funkcije $f_i(x), i = n,\ n-1,\ \ldots, 1$ prosleđuju funkciji $h(x)$ kao argumenti.

U JavaScript-u to može da izgleda ovako:

```
const g = n => n + 1;
const f = n => n * 2;
// compose :: a => x => y
const compose = (...fns) => x => fns.reduceRight((y, f) => f(y), x);

console.log(compose(f,g)(20));        //=> 42
const h1 = compose(f,g);
console.log(h1(20));            //=> 42
```

Ovde se koristi ugrađena metoda tipa `Array` pod imenom `reduceRight`[4] koja redukuje niz sa-desna-na levo, odnosno od dna ka vrhu.

Funkcija `pipe()` koja, takođe, koristi kuriranje je praktično upotrebljiva. Naime, pravilo je da se u JavaScript-u, pri ulančavanju metoda ulančane metode izvršavaju po redosledu koji je obrnut od njihovog navođenja u kodu. To je ljudima naporno za debagovanje pa je u [Eliot], predložen pomoćni program `pipe()` kojim se po redosledu navođenja (sa leva na desno, od vrha ka dnu) prati izvršavanje funkcija koje se komponuju. Korišćena je funkcija `reduce`[5] (redukuje niz sa leva na desno, odnosno od vrha ka dnu) koja je takođe ugrađena funkcija tipa `Array`. Kod izgleda ovako:

```
const trace = label => value => {
      console.log(`${ label }: ${ value }`);
      return value;
```

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

```
};

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

const g = n => n + 1;
const f = n => n * 2;

/*Primena funkcija se izvršava od vrha ka dnu (prvo f, zatim g): */
const h = pipe(g, trace('nakon g'), f, trace('nakon f'));
h (20);

/*
nakon g: 21
nakon f: 42
*/
```

Stvarna moć kuriranja i kuriranih funkcija je u tome što pojednostavljuju kompoziciju funkcija. Funkcija, kao što znamo, može da prihvati bilo koji broj ulaza, ali može vratiti samo jedan izlaz. Da bi funkcije bile kompozibilne, tip izlaza "prethodeće" funkcije mora biti usklađen sa očekivanim tipom ulaza funkcije koja sledi u lancu kompozicije. U stvari, to znači da u lancu kompozicije sve funkcije (osim prve) moraju da budu unarne – da prihvataju samo jedan argument.

Na primer, ako je data kompozicija:

```
f: a => b
g: b => c
h: a => c
```

i ako bi funkcija g bila takva da očekuje dva ulazna parametra, znači ovako:

```
f: a => b
g: (x, b) => c
h: a => c
```

oblik izlaza funkcije `f` ne bi bio u saglasnosti sa ulazom funkcije `g` pa kompozicija ne bi bila moguća.

Problem se rešava kuriranjem funkcije; u primeru bi se kurirala funkcija `g` što bi omogućilo da se "ubaci" problematični parametar `x`. Ista je situacija i sa pomoćnim programom `trace()` koji smo upravo pokazali: funkcija `trace()` prima dva ulaza (`label` i `value`) ali ih prima jedan po jedan. To omogućuje da se izvrši „inline" specijalizacija tako što će funkcija `pipe ()` da se pozove na sledeći način:

```
pipe(g, trace('nakon g'), f, trace('nakon f'))(20);
```

Ovde su pozivi `trace('nakon g')` i `trace('nakon f')` dve parcijalne aplikacije funkcije `trace()` koje vraćaju „specijalizovane" funkcije u kojima je `label` fiksirano (na vrednosti `'nakon g'` i `'nakon f'` respektivno) i koje očekuju da im se prosledi vrednost parametra `value.` Kada se te parcijalne aplikacije izvrše sa ulazom `value` (recimo 20), biće vraćene odgovarajuće povratne vrednosti (21 i 42 respektivno).

Postoji dobra praksa koja se zove „podaci na kraju" koja preporučuje da se u definiciji funkcije prvo navode parametri za specijalizaciju a na kraju podaci nad kojima se funkcija efektivno izvršava.

Ako je funkcija `trace()` nekurirana:

```
const trace = (label, value) => {
      console.log(`${ label }: ${ value }`);
      return value;
};
```

funkcija `pipe()` mora pozvati funkciju `trace()` sa dva argumenta što se može uraditi na sledeći način:

```
pipe (g, x => trace('nakon g', x), f, x => trace('nakon f', x))(20);
```

Ovakav poziv je nečitljiviji nego poziv u kome je funkcija `trace()` kurirana pa se prednost uvek daje kuriranju.

Na kraju, skrećemo požnju na još jednu stvar koja je čest izvor grešaka. Bez obzira da li je funkcija kurirana ili nije, mora se obezbediti da parametri pristižu u definisanom redosledu. Neka je funkcija `trace()` kurirana, ali takva da očekuje prvo `value` pa onda `label`:

```
const trace = value => label) => {
      console.log(`${ label }: ${ value }`);
      return value;
};
```

U tom slučaju, poziv

```
pipe (g, trace('nakon g'), f, trace('nakon f'))(20);
```

vratiće

21: nakon g

NaN: nakon f

Ispravan poziv funkcije `pipe()` mogao bi da izgleda ovako:

```
pipe (g, x => trace(x)('nakon g'), f, x => trace(x)('nakon f'))(20);
```

Kao što se iz izloženog može videti, kompozicija funkcija je uslovljena kuriranjem. To je potpuno u skladu sa činjenicom da je u λ-računu, koji je teorijska osnova funkcionalnog programiranja, funkcija u svom osnovnom obliku funkcija jedne promenljive, odnosno kurirana.

9.1.4.3. Kodiranje komponovanjem

U svojoj knjizi [Lonsdorf] Brajan Lonsdorf ima poglavlje pod istim ovakvim naslovom. Mi smo ga pozajmili jer ono u dve reči iskazuje suštinu funkcionalnog programiranja – u svojoj srži, funkcionalno programiranje je samo kompozicija/komponovanje funkcija. I ništa više.

U istom izvoru, Brajan Lonsdorf kompoziciju funkcija naziva „uzgajanjem funkcija“ gde je programer napredni uzgajivač koji bira jedinke (funkcije) za ukrštanje (komponovanje funkcija) i dobija nove, sposobnije jedinke (komponovane funkcije). Krajnji izlaz zavisi od raspoloživog „rasplodnog materijala“ i od sposobnosti „uzgajivača“ da odabere i „spari“ jedinke koje će dati najboljeg „potomka“ za rešavanje datog problema.

Mi smo već i pokazali neke implementacije komponovanja funkcija u JavaScript-u i njihovo korišćenje na jednostavnim zadacima. Ovde ćemo pokušati da malo dublje uđemo u koncept kompozicije funkcija i da ga uopštimo da bismo ga bolje razumeli.

Za početak, podsetićemo se osobine koja važi za svaku kompoziciju - asocijativnosti. U matematici to znači da je istinit iskaz:

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

U JavaScript-u to znači sledeće

```
const compose = (...fns) => x => fns.reduceRight((y, f) => f(y), x);
const g = n => n + 1;
const f = n => n * 2;
const h = n => n/2
let h1 = compose(f, compose(g, h))(10)    // 12
let h2 = compose(compose(f, g), h)(10)    // 12
```

Zatim ćemo prvi put uvesti neke pojmove iz drugog matematičkog temelja funkcionalnog programiranja – Teorije kategorija.

9.1.4.3.1. Komponovanje funkcija i teorija kategorija

Teorija kategorija je opšta teorija o matematičkim strukturama i njihovim relacijama. Primenljiva je na gotovo sve oblasti matematike. Moglo bi se kazati da je teorija kategorija najviši stepen apstrakcije u matematici koji omogućuje da se na uniforman način iskažu brojni parcijalni rezultati poput logike, teorije skupova, teorije homotopije, itd. Pored značaja za matematiku, teorija kategorija se sve šire primenjuje u praksi. Jedna njena primena je teorijska osnova jezika funkcionalnog programiranja.

Naravno, mi se nećemo upuštati u matematički rigorozum teorije kategorija, nego ćemo pokušati da na razumljiv način objasnimo njenu suštinu i da to ovde ilustrujemo na primeru kompozicije funkcija. U nastavku knjige celo poglavlje je posvećeno primeni teorije kategorija u funkcionalnom programiranju a primeri primene se pojavljuju na više mesta u knjizi.

U teoriji kategorija osnovni koncept je, naravno, ***kategorija***. Kategorija je kolekcija koja se sastoji od sledećih komponenti:

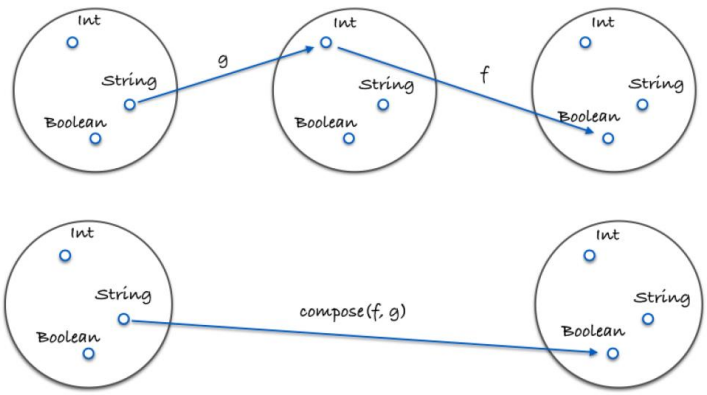
- Kolekcije objekata, koji mogu da budu bilo šta.
- Kolekcije morfizama koji mapiraju objekte i pri tome očuvavaju strukturu. Očuvanje strukture znači da „slike“ dva povezana objekta takođe moraju biti povezane.
- Operacije komponovanja morfizama, što znači da se dva morfizma `g` i `f`, gde morfizam `g` mapira objekat A na objekat B a morfizam `f` mapira objekat B na objekat C, mogu izraziti kao morfizam `h` koji mapira objekat A na objekat B.
- Posebnog morfizma koji se zove ***identitet***.

Iz aspekta naše trenutne teme – kompozicije funkcija – možemo ove komponente konkretizovati.

U našoj kolekciji objekata, objekti predstavljaju **tipove** (`Boolean` , `Number` , `Object` , itd).

Elementi naše kolekcije morfizama su **čiste funkcije**.

Naša operacija komponovanja morfizama je **funkcija `compose()`**. Sledeća slika (slika 8.1) ilustruje kompoziciju morfizama.



Slika 8.1 Kompozicija morfizama (funkcija)

Naš posebni morfizam koji se zove ***identitet*** je funkcija `id:: a → a`. To je funkcija koja prima neki ulaz i taj isti ulaz vraća kao svoj rezultat. Na prvi pogled, ne radi ništa korisno. Pokazuje se, međutim, da ona igra vrlo važnu ulogu u kompoziciji funkcija jer zadovoljava sledeći iskaz za sve unarne funkcije f:

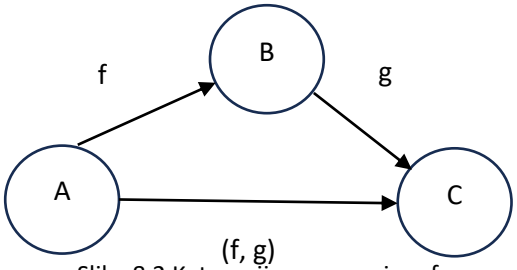
$$id \circ f = f \circ id = f.$$

To znači da funkcija `id` uvek može da se pojavi na mestu na kome želimo da predstavimo neku datu vrednost, pa i funkciju koja se u JavaScript-u tretira kao bilo koja vrednost što može da bude vrlo korisno pri pisanju koda point-free stilom.

Kategorija može biti raznih. Nas će interesovati kategorija koja predstavlja usmereni graf koji je osnovna apstrakcija u teoriji programskih jezika. U ovoj kategoriji, **objekti** su **čvorovi grafa**, **morfizmi** su **linkovi među čvorovima**, a **kompozicije morfizama** su **konkatenacije putanja među čvorovima**.

Slika 8.2 je vizualizacija kategorije usmerenog grafa.

Na ovoj slici, A, B i C su objekti (čvorovi grafa), `f` i `g` su linkovi među čvorovima a `(f, g)` je takođe link među čvorovima koji predstavlja konkatenaciju putanja A → B i B →C.



Slika 8.2 Kategorija usmereni graf

9.1.4.3.2. Primena komponovanja funkcija

Zaključićemo ovaj odeljak izlaganjem primera izrade i komponovanja funkcija kojim želimo da ilustrujemo mogućnosti primene funkcija i komponovanja funkcija zarad ekonomične izrade kvalitetnog i čitljivog softvera pogodnog za održavanje.

Započećemo nabrajanjem karakteristika koje svaka apstrakcija, pa i funkcija, treba da zadovolji:

- kompozibilnost,
- ponovna upotrebljivost,
- nezavisnost,
- konciznost, i
- jednostavnost.

Kompozibilnost se odnosi na implementaciju funkcije na način kojim se omogućuje njeno jednostavno uklapanje u lanac kompozicije funkcija. Tehnički, to znači da funkcija treba da bude napisana u kuriranoj formi što je u jeziku JavaScript omogućeno.

Ponovna upotrebljivost je jedan od osnovnih principa izrade softvera. Po svojoj prirodi funkcije su inherentno ponovno upotrebljive – one se i pišu sa namerom višekratnog pozivanja. Stoga se ovde ponovna upotrebljivost odnosi na potrebu korišćenja implementirane funkcionalnosti (na primer, da li će se funkcija pozivati u velikom broju aplikacija više stotina/hiljada puta) i tehničku implementaciju koja pogoduje ponovnoj upotrebljivosti (na primer, korišćenje funkcije višeg reda kojoj se može kao argument proslediti druga funkcija). Dobar primer u jezicima funkcionalnog programiranja a i u JavaScript-u je funkcija `reduce()` koja se odlikuje visokom ponovnom upotrebljivošću ako je implementirana tako da joj se transformacija prosleđuje kao funkcija. Sama funkcija ima izuzetno visoku ponovnu upotrebljivost jer se može iskoristiti za rešavanje široke klase zadataka pa čak i za modeliranje drugih apstrakcija kao što su `map()` i `filter()` koje se i same odlikuju visokom ponovnom upotrebljivošću. Pomenute tri funkcije, koje predstavljaju operacije nad listama (uređenim kolekcijama vrednosti), izuzetno su značajne za funkcionalno programiranje. Zbog toga

ćemo im u nastavku knjige posvetiti poseban odeljak u kome ćemo ih detaljno objasniti. Ovde samo da vrlo sažeto kažemo šta koja od njih radi: `reduce()` transformiše ulaznu listu u jednu vrednost, `map()` transformiše ulaznu listu u novu listu iste dimenzije kao ulazna, a `filter()` iz ulazne liste izdvaja u novu listu elemente koji zadovoljavaju zadati uslov.

Nezavisnost funkcije znači da funkcija svoj zadatak rešava tako da ga ostvari uz što manje interakcije sa drugim, spoljašnjim komponentama. Gledano iz praktičnog, programerskog ugla to znači da je poželjno da funkcija bude "samodovoljna" i da se njena interakcija sa drugim komponentama ostvaruje putem ulaznih parametara i vraćene vrednosti. Podrška za funkcije višeg reda i komponovanje funkcija u JavaScript-u pogoduju postizanju ove karakteristike.

Konciznost se odnosi na pisanje sažetog koda koji sadrži minimum iskaza potreban za ostvarivanje nameravane funkcionalnosti. Ovo je prilično kontroverzna karakteristika jer može da rezultuje kodom koji je težak za čitanje i održavanje. Funkcija tu pomaže mnogo jer omogućuje deklarativni stil programiranja koji je vrlo pogodan za pisanje konciznog koda lakog za čitanje i razumevanje, što se u velikoj meri može postići i u JavaScript-u.

Jednostavnost se odnosi na kapacitet apstrakcije da doprinese primeni Zakona jednostavnosti u softverskom inženjerstvu koji prema izvoru [Max Kanat-Alexander] kaže: " Lakoća održavanja bilo kog softvera je proporcionalna jednostavnosti njegovih pojedinačnih delova." Ono što se iz ovog iskaza može uočiti je da on govori o jednostavnosti pojedinačnih delova softvera, ne o jednostavnosti ukupnog softvera. To znači da se podrazumeva da će softver biti skup povezanih delova, a ne monolitna celina. Funkcionalno programiranje je stil programiranja koji softver posmatra baš kao povezane komponente, pri čemu su komponente jednostavne funkcije a povezivanja se ostvaruje putem jednoznačnog i jednostavnog mehanizma komponovanja funkcija. JavaScript funkcije tretira funkcije kao "građane prvog reda" i omogućuje njihovo komponovanje.

9.1.4.3.2.1. Ulančavanje

Jedna od stvari koja je od izuzetnog značaja pri komponovanju funkcija je tip ulaza i tip povratne vrednosti. Iako to nije neophodno, komponovanje funkcija mnogo je "lagodnije" ako je tip povratne vrednosti funkcije u saglasnosti sa tipom ulaza funkcije koja će da ga primi. U takvoj situaciji izbegava se provera tipa u funkciji koja ga prima – prosto se pretpostavlja da je on odgovarajući. Takav način komponovanja, koji se naziva ulančavanje funkcija, već smo videli u nekim primerima a ovde ćemo ga detaljnije objasniti i ilustrovati primerima.

Za početak ćemo da vizualizujemo konceptualni tok podataka koji je univerzalno primenljiv na ulančavanje funkcija:

$$\texttt{IzlaznaVrednost} \leftarrow f_n \leftarrow f_{n-1} \leftarrow f_{n-2} \leftarrow \ldots \leftarrow f_1 \leftarrow \texttt{UlaznaVrednost}$$

U ovoj ilustraciji komponuju se funkcije f_i, $i=1,n$. `UlaznaVrednost` je ulaz u funkciju f_1, a `IzlaznaVrednost` je rezultat koji se dobija na izlazu funkcije f_n. Naravno, svaka od funkcija f_i, $i=1,n-1$ vraća svoj rezultat koji je ulaz u funkciju f_{i+1} . Kao što rekosmo, bilo bi dobro da izlaz iz funkcije f_i, $i=1,n-1$ odgovara ulazu u funkciju f_i, $i=2,n$. Da vidimo kako to izgleda na jednostavnom primeru preuzetom iz [Simpson].

Primer 1

U ovom primeru se iz stringa koji sadrži reči (podstringovi razdvojeni znakom blanko ili znakom beksleš ****) formira niz čiji elementi su reči iz zadatog stringa uz zahtev da svaka reč počinje malim slovom i da niz ne sadrži ponovljene reči. Evo koda koji to radi.

```
function reci(str) {
      return String( str )
            .toLowerCase()
            .split( /\s|\b/ )
            .filter( function alpha(v){
```

```
                return /^[\w]+$/.test( v );
            } );
}
function jedinstveno(list) {
      var jedinstvenaLista = [];
      for (let v of list) {
      // Vrednost još nije u novoj listi?
            if (jedinstvenaLista.indexOf( v ) === -1 ) {
                  jedinstvenaLista.push( v );
            }
      }
      return jedinstvenaLista;
}
var nasTekst = "Za kompoziciju dveju funkcija, proslediti \
izlaz poziva prve funkcije na ulaz  \
pozivu druge funkcije.";
var nadjeneReci = reci( nasTekst );
var korisceneReci = jedinstveno( nadjeneReci );
korisceneReci;

['za', 'kompoziciju', 'dveju', 'funkcija', 'proslediti', 'izlaz', 'poziva',
'prve', 'funkcije', 'na', 'ulaz', 'pozivu', 'druge']
```

U ovom kodu kompozicija funkcija se javlja u funkciji `reci ()` u kojoj se formira niz koji sadrži reči prepoznate u stringu. Ovde se ulančavaju redom sledeće funkcije: `filter()`, `split()` i `toLowerCase()`. Zatim se poziva druga funkcija `jedinstveno()` koja eliminiše duplikate reči kojoj se na ulaz postavlja vrednost koju je vratila funkcija `reci ()`.

To znači da će naš kod da radi i ako komponujemo funkcije `reci()` i `jedinstveno()` na sledeći način:

```
var korisceneReci = jedinstveno( reci(nasTekst)) );
```

Literatura uz Poglavlje 9

1. Brian Lonsdorf, Professor Frisby's Mostly Adequate Guide to Functional Programming, Samizdat, 2019.
2. Eric Elliot, Composing Software, Leanpub, 2019
3. Michael Fogus, Functional JavaScript, O'Reilly Media, Inc., 2013.
4. James Sinclair, HOW TO DEAL WITH DIRTY SIDE EFFECTS IN YOUR PURE FUNCTIONAL JAVASCRIPT, 2018, dostupno na https://jrsinclair.com/articles/2018/how-to-deal-with-dirty-side-effects-in-your-pure-functional-javascript/
5. Max Kanat-Alexander, Code Simplicity, O'Reilly Media, Inc., 2012.

Poglavlje 10: REKURZIJA

U knjizi "The Dao of Functional Programming", Bartoš Milevski objašnjava rekurziju poznatim fenomenom "beskonačnog ogledala" : "Kada zakoračite između dva ogledala, vidite svoj odraz, odraz svog odraza, odraz odrazovog odraza, itd. Svaki odraz je definisan prethodnim odrazom, a zajedno proizvode utisak beskonačnosti."

Drugi primer koji je manje apstraktan i pomaže da se razume sama suština rekurzije a i način na koji se ona efektivno izvršava jeste red ljudi koji čekaju na neku uslugu – na primer da kupe kartu za pozorišnu predstavu. Ako stojite u takvom redu, odmah Vam pada na pamet pitanje: Kada ću ja doći do blagajne, odnosno koliko ljudi ima u redu ispred mene?

Jedan način da to saznate je de se izmaknete iz reda i prebrojite ljude koji čekaju u redu. To je rešenje koje odgovara dobroj, staroj petlji u imperativnom programiranju. I zahteva od Vas da uložite dodatni napor – da brojite.

Ako baš i niste skloni dodatnim naporima, možete to da uradite i na drugi način – rekurzivno. Taj drugi način je da pitate osobu koja je neposredno ispred Vas u redu koliko je ljudi u redu ispred nje i da, kada Vam ona kaže broj osoba koje su ispred nje, Vi na taj broj dodate 1. Kako (po pretpostavci) ni osoba ispred Vas ne zna koja je po redu, ona može da uradi isto što ste i Vi uradili – da pita osobu koja je u redu neposredno ispred nje koja je po redu i kada dobije odgovor da doda 1. Taj postupak ponavlja svaka osoba u redu dok se ne stigne do osobe koja je trenutno na blagajni – nema nikog ispred sebe. Ona zna da je broj ljudi ispred nje 0 i to će da saopšti osobi koja je neposredno iza nje. Sada ta (druga u redu) osoba može da doda 1 na 0 i da osobi koja je neposredno iza nje saopšti broj 1. Postupak se ponavlja za svaku osobu u redu dok se ne stigne do poslednje osobe u redu, to jest Vas. Vi sada znate koliko je osoba u redu ispred osobe koja je neposredno ispred Vas (ona Vam je to kazala) i kada na broj koji Vam je saopšten dodate 1 znaćete koliko njih još treba da kupi kartu pre nego što Vi stignete na red.

U ovom poglavlju bavićemo se rekurzijom a pomenućemo i dualnu operaciju zvanu ko-rekurzija.

https://en.wikipedia.org/wiki/Recursion_(computer_science)

https://www.codingame.com/playgrounds/2980/practical-introduction-to-functional-programming-with-js/recursion

10.1. O rekurziji u matematici i programiranju

U matematici i računarskoj nauci, **rekurzija** je pojava kada **definicija koncepta** ili **procesa zavisi** od **jednostavnije ili prethodne verzije samog sebe**. U programiranju to znači da se funkcija koja se definiše poziva unutar same sebe - sopstvene definicije. Iako se na taj način očigledno definiše beskonačan broj instanci (vrednosti funkcije), to se u praksi (kodiranju) radi tako da se ne može pojaviti beskonačna petlja ili beskonačan lanac referenci.

10.1.1. Osnovno o rekurziji

Rekurzivna definicija funkcije definiše vrednosti funkcije za neke ulaze putem vrednosti iste te funkcije za druge (obično „manje") ulaze. Teorema rekurzije garantuje da takva definicija zaista definiše funkciju koja je jedinstvena.

Teorema o rekurziji: Za zadati skup X, element a skupa X i funkciju $f: X \to X$, postoji jedinstvena funkcija

$$F(0) = a$$

$$F(n+1) = f\big(F(n)\big)$$

za svaki prirodan broj n.

Formalno se rekurzija u matematici i računarskoj nauci definiše na sledeći način.
Klasa objekata ili metoda ispoljava rekurzivno ponašanje ako se može definisati putem dva svojstva:

- **Bazni slučaj** (ili slučajevi) – terminirajući scenario koji ne koristi rekurziju za proizvođenje rezultata;
- **Rekurzivni korak** —skup pravila koji sve sukcesivne slučajeve vode ka baznom slučaju.

Primer 1: ***Brojanje unazad***
Bazni slučaj: **n=0**
Rekurzivni korak: **n - 1**

```
let brojiUnazadOd = (n) => {
    if (n === 0) { // bazni slučaj
        console.log("IZBROJAO!");
    }
    else {
        console.log(n);
        brojiUnazadOd (n-1); // REKURZIVNI POZIV ZA n-1
    }
}

brojiUnazadOd(5);
```

Primer 2: **Faktorijel**
Bazni slučaj: **x = 0**
Rekurzivni korak: **x! = (x-1)!*x**

```
// program za sračunavanje faktorijela
function faktorijel(x) {

    if (x === 0) { // Bazni uslov x = 0
        return 1;
    }

    // Ako je argument pozitivan broj
    else {
        return x * faktorijel(x - 1); // rekurzivni poziv
    }
}

const broj = 3;

// pozivanje funkcije faktorijel() ako je broj nenegativan
if (broj > 0) {
    let rezultat = faktorijel(broj);
    console.log(`${broj}! = ${rezultat}`);
}
```

Primer 3: Fibonačijev niz
Bazni slučaj 1: $\boldsymbol{F(0) = 0}$
Bazni slučaj 2: $\boldsymbol{F(1) = 1}$
Rekurzivni korak: $\boldsymbol{F(n) = F(n-1) + F(n-2)}$ **za celobrojno** $\boldsymbol{n > 1}$.

```
function fib ( n ){
       if ( n < 2 )
```

```
            return n;
        else
            return fib(n - 1) + fib(n - 2);
        }
console.log(fib(5)) // 5
```

Primer 4: ***porodično stablo***
Bazni slučaj: **nema roditelja**
Rekurzivni korak: **roditeljev roditelj**

```
let mojaFamilija =[
  { "id": "Deda", "roditelj": null },
  { "id": "Otac", "roditelj": "Deda" },
  { "id": "Ja", "roditelj": "Otac" },
  { "id": "Sin", "roditelj": "Ja" },
  { "id": "Kćerka", "roditelj": "Ja" },
  { "id": "Brat", "roditelj": "Otac" },
  { "id": "Bratić", "roditelj": "Brat" },
  { "id": "Bratičina", "roditelj": "Brat" },
  { "id": "Sestra", "roditelj": "Otac" },
  { "id": "Stric", "roditelj": "Deda" },
  { "id": "Stričević", "roditelj": "Stric" }
]

const porodicnoStablo = (familija, roditelj) => {
      return familija
      .filter(p => p.roditelj === roditelj)
      .reduce((deca, dete) => {
               deca[dete.id] = porodicnoStablo(familija, dete.id); /*
                               Rekurzivni poziv */
            return deca;
      },{}); // Bazni uslov je prazan objekat
}
porodicnoStablo(mojaFamilija,null)porodicnoStablo(mojaFamilija,null)
```

10.1.2. Osnovno o ko-rekurziji

Ko-rekurzija je vrsta operacije koja je dualna rekurziji.

Za razliku od rekurzije koja radi analitički, počinjući od podataka koji su „dalje“ od baznog slučaja i ponavljajući rekurzivne korake sve dok se ne dođe do baznog slučaja, ko-rekurzija radi sintetički: počinje od baznog slučaja i proizvodi podatke ponavljanjem rekurzivnih koraka „udaljavajući se“ od baznog slučaja. Suština ko-rekurzivnih algoritama je da oni koriste podatke koje sami proizvode, deo po deo, kada postanu dostupni i potrebni, da bi proizveli nove delove podataka. Sledi primer porodičnog stabla kao ilustracija ko-rekurzije.

Tamo gde rekurzija omogućuje da se radi na proizvoljno složenim podacima, sve dok se oni mogu svesti na bazne slučajeve, ko-rekurzija omogućuje da se proizvode proizvoljno složene i potencijalno beskonačne strukture podataka, kao što su tokovi (*stream*), sve dok se mogu proizvesti iz baznih slučajeva u nizu konačnih koraka.
Iako ko-rekurzija počinje od osnovnog stanja i proizvodi sledeće korake deterministički, i ona može da se odvija beskonačno, ili može da troši više nego što proizvodi i tako postaje *neproduktivna*.

Ko-rekurzija može da proizvede i konačne i beskonačne strukture podataka kao rezultate i može da koristi samoreferentne[6] strukture podataka.
Često se koristi u sprezi sa lenjom evaluacijom da se proizvede konačan (potreban) podskup potencijalno beskonačne strukture. Ko-rekurzija je posebno važan koncept u funkcionalnom programiranju, gde ko-rekurzija i ko-podaci (*codata*, *co-data*)[7] omogućavaju totalnim jezicima[8] da rade sa beskonačnim strukturama podataka.
Mnoge funkcije koje se tradicionalno posmatraju kao rekurzivne mogu se, čak prirodnije, posmatrati kao ko-rekurzivne funkcije koje se završavaju u datoj fazi. Primer su rekurentne relacije poput Fibonačijevog niza i faktorijela.
Primer 5: Faktorijel ko-rekurzivno
Rekurzivna definicija **0! = 1, n! = n*(n-1)**

Da bi rekurzivno izračunala svoj rezultat za dati ulaz, rekurzivna funkcija poziva samu sebe (u stvari, svoju kopiju) sa drugačijim (na neki način „manjim“) ulazom i koristi rezultat ovog poziva da konstruiše svoj rezultat. Rekurzivni poziv radi to isto, osim ako nije dostignut bazni slučaj. Tako se u procesu razvija stek poziva. Na primer, da bi se izračunao $fac(3)$, rekurzivno se poziva $fac(2)$, $fac(1)$, $fac(0)$ („namotavanje“ steka), u kom trenutku se rekurzija završava sa $fac(0) = 1$, a zatim se stek „odmotava“ obrnutim redosledom i rezultati se izračunavaju na povratku duž steka poziva do početnog okvira poziva $fac(3)$ koji koristi rezultat $fac(2) = 2$ za izračunavanje konačnog rezultata kao $3 \times 2 = 3 \times fac(2) =: fac(3)$ i konačno vrati $fac(3) = 6$. To je potpuno analogno sa primerom ljudi koji čekaju u redu da kupe kartu za pozorište: „namotavanje“ steka je postavljanje pitanja čoveku koji je neposredno ispred, „odmotavanje“ steka je prijem odgovora, dodavanje jedinice i saopštavanje te vrednosti čoveku koji je neposredno iza. U ovom primeru funkcija vraća jednu vrednost.
Ovo odmotavanje steka se može eksplicirati korekurzivnim definisanjem faktorijel kao iteratora, gde se počinje sa slučajem $1 = 0!$, zatim iz ove početne vrednosti konstruišu faktorijalne vrednosti za rastuće brojeve 1, 2, 3... kao u gornjoj rekurzivnoj definiciji sa „vremenskom strelicom“ koja je obrnuta, čitajući je unazad kao $n! \times (n+1) =: (n+1)!$. Tako definisan korekurzivni algoritam proizvodi tok svih faktorijala. Ovo se može konkretno primeniti kao generator. Simbolički, imajući u vidu da izračunavanje sledeće faktorijel vrednosti zahteva praćenje i n i f (f je prethodna faktorijel vrednost), ovo se može predstaviti kao:

$$n, f = (0,1): \big(n+1, f \times (n+1)\big)$$

Za naš primer čekanja u redu to bi bilo kao da nema postavljanja pitanja, nego ljudi u redu sponatno (bez pitanja) saopštavaju svoju poziciju onome koji sledi neposredno.

Primer 6: Fibonačijev niz ko-rekurzivno

10.2. Rekurzija u programiranju

U svojoj čuvenoj knjizi „Algorithms + Data Structures = Programs“ , Niklaus Virt (Niklaus Emil Wirth)[9] rekao je: “ Snaga rekurzije očigledno leži u mogućnosti definisanja beskonačnog skupa objekata pomoću konačnog iskaza. Na isti način, beskonačan broj proračuna može se opisati pomoću konačnog rekurzivnog programa, čak i ako ovaj program ne sadrži eksplicitna ponavljanja.“

[6] Samo-referenca je koncept koji uključuje upućivanje na sebe ili na sopstvene atribute, karakteristike ili akcije.

[7] Ko-podaci su ko-induktivno definisani tipovi, tipično beskonačne strukture podataka kao što su tokovi (strimovi); Ko-indukcija je matematički dual strukturalnoj indukciji. Strukturalna indukcija je metod dokazivanja koji predstavlja generalizaciju matematičke indukcije nad prirodnim brojevima.

[8] Totalno funkcionalno programiranje je programska paradigma koja ograničava opseg programa na one koji dokazivo terminiraju.

[9] https://en.wikipedia.org/wiki/Niklaus_Wirth

Većina programskih jezika podržava rekurziju a neki funkcionalni programski jezici (na primer, Clojure) uopšte nemaju konstrukte petlje, već repetitivne korake izvršavaju isključivo putem rekurzije. Dokazano je da su rekurzivni jezici Tjuring-kompletni što znači omogućuju sve ono što omogućuju imperativni jezici.

Rekurzija je generalno manje efikasna od iteriranja. Glavni razlozi za to su što pozivanje funkcije troši više CPU resursa i što svaki poziv dodaje novi frejm na stek što znači trošenje memorijskih resursa koji su ograničeni pa može da dođe do "ispadanja iz memorije". Ipak, rekurzija se smatra jednim od najvažnijih koncepata u programiranju jer se u potpunosti poklapa sa strategijom rešavanja problema putem dekomponovanja na manje probleme (poželjno istog tipa), rešavanje tih manjih problemi i kombinovanje rešenja u konačan rezultat. Prednosti rekurzije su što ona smanjuje jaz između elegancije i kompleksnosti u programu,

10.2.1. Rekurzivne funkcije i algoritmi

Uobičajena taktika pravljenja algoritama je da se problem podeli na podprobleme istog tipa kao originalni, reše se ti podproblemi i rešenja kombinuju u konačan rezultat. Kada se ta taktika kombinuje sa tabelom koja čuva rezultate rešenja podproblema da bi se izbeglo njihovo ponovljeno rešavanje i trošenje resursa, često se naziva *dinamičkim programiranjem* ili *memoizacijom*.

10.2.1.1. Tipovi rekurzije

Rekurzija može da se pojavi u više oblika koji se razlikuju u detaljima. Stoga se može izvršiti klasifikacija rekurzije. Iako klasifikacije na koje se nailazi u literaturi nisu potpuno identične niti su im klase disjunktne, najrasprostranjenije klasifikacije rekurziju klasifikuju na osnovu dva kriterijuma: (1) broju samo-referenciranja i (2) načina pozivanja rekurzivne funkcije.

U odnosu na broj samo-referenciranja razlikuju se:

- **Pojedinačna rekurzija** u kojoj se javlja **samo jedno samo-referenciranje**.
- **Višestruka rekurzija** u kojoj javlja **više od jednog samo-referenciranja**.

U odnosu na način pozivanja rekurzivne funkcije razlikuju se:

- **Direktna rekurzija** u kojoj **rekurzivna funkcija poziva samu sebe**.
- **Indirektna/uzajamna rekurzija** u kojoj rekurzivna funkcija ne poziva samu sebe, već **rekurzivnu funkciju poziva druga funkcija pozvana od strane rekurzivne funkcije** (direktno ili indirektno). Odnosno, situacija u kojoj rekurzivna funkcija poziva ne-rekurzivnu funkciju koja, zauzvrat, poziva rekurzivnu funkciju koja je nju pozvala

U nastavku slede jednostavni primeri koji ilustruju različite tipove rekurzije.

```
 /*
OVAKO NAM IZGLEDA FAJL SISTEM

    home
    ├── andrea
    │   ├── funds.csv
    │   └── paper.pdf
    ├── john
    │   ├── logs
    │   │   ├── logs1
    │   │   ├── logs2
    │   │   ├── logs3
    │   │   └── logs4
```

```
    │   ├── summer1.jpg
    │   ├── summer2.jpg
    │   └── summer3.jpg
    ├── notes.txt
    └── todo.txt

*/

const tree = {
    name : "home",
    files : ["notes.txt","todo.txt"],
    subFolders: [{
        name : "andrea",
        files : ["paper.pdf","funds.csv"],
        subFolders: []
    },
    {
        name: "john",
        files : ["summer1.jpg","summer2.jpg", "summer3.jpg"],
        subFolders: [{
            name : "logs",
            files : ["logs1","logs2","logs3","logs4"],
            subFolders: []
        }]
    }]
};
find = element => tree =>
         {
            if (tree.files.indexOf(element)!==-1){ /* Ovo važi ako
                                                   je element u tom folderu */
               return true;
            }
            else if (tree.subFolders.length !== 0){ /* Ovo važi
                                                      ako ima pod-foldera */
                const otherFolders = tree.subFolders.map(find(element));
/*                traži element u svakom pod-folderu
                      const aOrB = (a,b)=>a || b; /* funkcija koja
                                                  primenjuje or operator */
                      const found = otherFolders.reduce(aOrB, false); /*
                      vraća true ako je našao element u najmanje jednom
                      pod-folderu */
                      return found;
                   }else{
                      return false; /* vraća false ako element nije u
                            tom folderu i taj folder nema pod-foldera */
                   }
                };

console.log("paper.pdf: "+find("paper.pdf")(tree));
console.log("randomfile: "+find("randomfile")(tree));

let mojaLista =[
  { "id": "Grandad", "parent": null },
  { "id": "Dad", "parent": "Grandad" },
```

```
  { "id": "You", "parent": "Dad" },
  { "id": "Son", "parent": "You" },
  { "id": "Daughter", "parent": "You" },
  { "id": "Brother", "parent": "Dad" },
  { "id": "Nephew", "parent": "Brother" },
  { "id": "Niece", "parent": "Brother" },
  { "id": "Sister", "parent": "Dad" },
  { "id": "Uncle", "parent": "Grandad" },
  { "id": "Cousin", "parent": "Uncle" }
]
const buildTree = (list, parent) => {
      return list
      .filter(p => p.parent === parent)
      .reduce((children, child) => {
            children[child.id] = buildTree(list, child.id);
            return children;
      },{});
}
buildTree(mojaLista,parent)

// da se dobije
"Grandad": {
  "Dad": {
    "You": {
      "Son": {},
      "Daughter": {}
    },
    "Brother": {
      "Nephew": {},
      "Niece": {}
    },
    "Sister": {}
  },
  "Uncle": {
    "Cousin": {}
  }
}
}
```

Literatura uz Poglavlje 10