Univerzitet Singidunum
www.singidunum.ac.rs

Funkcionalno programiranje

Školska 2024/25 godina

Letnji semestar

Tema 3: Funkcije višeg reda

Sadržaj

- Šta se sve može sa funkcijama u JavaScript-u
- Šta je funkcija višeg reda
- Parcijalna aplikacija
- Kuriranje

Šta se sve može sa funkcijama

- Baš kao i tipovi `number`, `string`, ili `object`, funkcije mogu da se:
  - Dodele kao vrednost identifikatoru (varijabli)
  - Dodele vrednostima svojstava objekta
  - Prosleđuju kao argumenti
  - Vraćaju kao povratne vrednosti iz funkcije

Dodela funkcije kao vrednosti

- Funkcije nisu ništa drugo do podaci.
- Zbog toga se mogu skladištiti u varijable:

```
let fn = () => {alert (" Zdravo, ja sam fn ")}

// Vrednost varijable fn je kod funkcije
alert (" fn sadrži: " + fn)
// Ispis: let fn = () => {alert (" Zdravo, ja sam fn ")}

// Tip onoga na šta pokazuje fn je function
alert ( "Tip onoga na šta pokazuje fn je: " + typeof fn)
// A ovo je poziv funkcije
fn();  // Ispis: Zdravo, ja sam fn
```

Prosleđivanje argumenata: može i funkcija

```
let tellType = (arg) => {
   console.log(" Tip argumenta koji sam primio je " +
   typeof arg)
}

let jedan = 1
tellType (jedan)

let object = {
   a : "Ja sam obeležje a ",
   b : tellType }

tellType (object)
tellType (object.a)
tellType (object.b)
```

Vraćena vrednost: može i funkcija

- Funkcija kao rezultat svog izvršavanja može da vrati drugu funkciju

```
// Primer:
// String je ugrađena JS funkcija koja kreira string
console.log (' Tip promenljive String je: ' + typeof String)
console.log (' Promenljiva String pokazuje na: ' + String)
console.log ("Poziv String ('HOC') vraca: " + String ('HOC') )

/* Sada pravimo funkciju crazy () koja vraća ugrađenu funkciju
String */
let crazy = () => { return String }
console.log (' Tip promenljive crazy je: ' + typeof crazy)
console.log (' Promenljiva crazy pokazuje na: ' + crazy)
console.log ("Poziv crazy ('HOC') vraca: " + crazy ('HOC') )
console.log ("Poziv crazy()('HOC') vraca: " + crazy ()('HOC'))
```

Funkcija višeg reda

- Funkcija višeg reda je funkcija koja:
  - Može da primi drugu funkciju kao argument,
  - Može da vrati drugu funkciju kao izlaz,
  - Može i jedno i drugo od navedenog

Zašto su važne funkcije višeg reda

- Kompozicija funkcija je temelj FP-a
- Funkcije višeg reda su temelj kompozicije funkcija
  - Bez kompozicije nema FP-a
  - Bez funkcija višeg reda nema kompozicije
- Funkcije višeg reda značajno olakšavaju apstrakciju.
  - Na primer, možemo da kreiramo funkciju koja apstrahuje iteriranje nad listom i akumuliranje vraćene vrednosti tako što će se proslediti **fukcija** koja rukuje *delovima koji su različiti.*

Zašto su moguće funkcije višeg reda

- Zato što u funkcionalnom programiranju funkciju gledamo na isti način kao što je gleda matematika
- A matematika kaže da se funkcije mogu komponovati:
  - Ako imamo dve funkcije: $f(x)$ i $g(x)$, legalno je da se napiše sledeće:

$$y = f(x)$$

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

Matematika: funkcije više promenljivih

- Matematika poznaje funkcije koje zavise od više argumenata:

$$f(x, y, z) = 256x + 48y - 23z$$

- Kako se sračunava vrednost funkcije za $x = 1, y = 2, z = 3$?
  - Može ovako (Prvi slučaj):

$$f(1, 2, 3) = 256 * 1 + 48 * 2 - 23 * 3$$

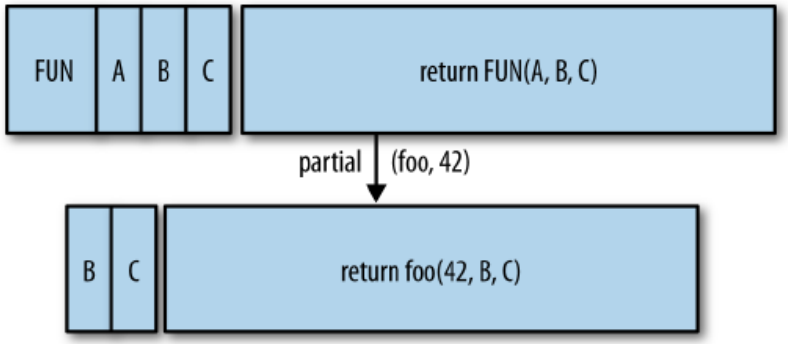
  - A može i ovako (Drugi slučaj):

$$f(1, 2, z) = 256 * 1 + 48 * 2 - 23z = g(z) = 352 - 23z$$
$$g(3) = 352 - 23 * 3 = 283$$

- Mi, u stvari, u funkciji zamenjujemo "promenljive" njihovim vrednostima.
  - U prvom slučaju "odjednom".
  - U drugom slučaju prvo dve a zatim treću pri čemu u svakom koraku primenjujemo različite funkcije koje nastaju kao rezultat svake zamene. Na kraju nema ništa da se zameni i funkcija je sračunata.

Drugi slučaj: Parcijalna aplikacija

- Ako funkcija ima više argumenata, može se pojaviti potreba da se neki od njih specificiraju ranije, a da se specificiranje drugih ostavi za kasnije.
- Ovaj koncept ima izuzetno značajnu ulogu u funkcionalnom programiranju i zove se ***pracijalna aplikacija***.
- Parcijalno primenjena funkcija je funkcija koja je "parcijalno" izvršena i spremna je za sledeće izvršavanje čim se zadaju njeni preostali argumenti.



Parcijalna aplikacija: primer$_1$

- Posmatra se sledeća funkcija:

```
function ajax(url,data,callback) {
// ..
}
```

- Potrebno je napraviti nekoliko API poziva gde su URL-i poziva poznati unapred (parametar `url`), dok će podaci (parametar `data`) i povratni poziv za rukovanje odgovorom (parametar `callback`) biti poznati tek kasnije.
- Pretpostavimo da se ti podaci odnose na kupca (`person`) i narudžbu (`order`).

Parcijalna aplikacija: primer$_2$

- Dve su mogućnosti:
  - Pozvati funkciju `ajax(..)` tek kada je sve poznato a do tada referisati neke globalne konstante za URL. Međutim, to nije baš mnogo lep kod – globalne konstante baš i nisu u duhu funkcionalnog programiranja.
  - Drugi, čistiji način je da se naprave nove funkcije koje i dalje pozivaju `ajax(..)`, i manuelno postavljaju prvi argument na URL API-a od interesa, a čekaju da prihvate druga dva argumenta kasnije.

Parcijalna aplikacija: primer$_3$

- Ako primenjujemo drugi pristup, funkcije mogu da izgledaju ovako:

```
function getPerson(data,cb) {
ajax( "http://some.api/person", data, cb );
  // parcijalno primenjena funkcija ajax()
}

function getOrder(data,cb) {
ajax( "http://some.api/order", data, cb );
  // parcijalno primenjena funkcija ajax()
}
```

Parcijalna aplikacija: primer$_4$

- Sada treba naći mudar način da se specificiraju omotači koji će omogućiti pozivanje tih funkcija.
- Osnovni princip pri pravljenju softvera je da pokušamo da napravimo nešto što je višekratno upotrebljivo tako što će se uočiti obrasci gde se iste stvari rade repetitivno.
- Da vidimo, dakle, šta je obrazac u našem primeru

Parcijalna aplikacija: obrazac

- Funkcije `getOrder(data,cb)` i `getPerson(data,cb)` su *parcijalne aplikacije* funkcije ajax(url,data,cb).
- Ovde se primenjuju samo neki argumenti na početku - specifično argument za `url` parametar - dok se ostali ostavljaju za kasniju primenu.
- Formalno gledano ***parcijalna aplikacija je striktno redukcija arnosti funkcije***; ovde se redukuje orignalna arnost funkcije `ajax()` sa 3 na 2 za svaku od funkciju `getOrder()` i `getPerson()`.

Parcijalna aplikacija: funkcija `partial`

```
function partial (fn,...presetArgs) {
    return function partiallyApplied (...laterArgs) {
        return fn( ...presetArgs, ...laterArgs );
    };
}
```

- Funkcija `partial()` prima funkciju `fn` za koju se vrši parcijalna primena.
- Zatim se svi argumenti koji se prosleđuju funkciji `fn` objedinjuju u niz `presetArgs` i sačuvavaju za kasnije.
- Kreira se nova unutrašnja funkcija (`partiallyApplied()`) i ta funkcija se vraća.
- Pri tome, argumenti unutrašnje funkcije se objedinjuju u niz `laterArgs`.

Parcijalna aplikacija: uloga zatvaranja

- U unutrašnjoj funkciji `partiallyApplied()` referenciraju se `fn` i `presetArgs` .
- Kako to radi? Nakon što `partial()` završi izvršavanje, kako unutrašnja funkcija ostaje u stanju da može da referencira `fn` i `presetArgs`?
- Odgovor je ***zatvaranje***!
  - Unutrašnja funkcija `partiallyApplied(..)` **zatvara i nad `fn` i nad `presetArgs` varijablama**, tako da može da im pristupi kasnije, bez obzira gde se funkcija izvršava.
- To je razlog zbog koga je razumevanje zatvaranja od kritičnog značaja u funkcionalnom (i ne samo funkcionalnom) programiranju!

Funkcija `partial()`: primer

```
function fn (x, y, z){
    return x + y + z
        }
let x=1, y=2, z=3;
const fnPartial1 = partial (fn, x) // NaN
const fnPartial2 = partial (fnPartial1, y) //NaN
const fnPartial3 = partial (fnPartial2,z) // 6
let a = 12
const fnPartial4 = partial (fnPartial2,a) // 15
const fnPartial5 = partial (fnPartial1,y,a) // 15
```

Metoda **`bind`** i parcijalne funkcije

- Metoda `bind` može da fiksira ne samo `this`, već i parametre funkcije. Potpuna sintaksa je:

`func.bind(context, [arg1], [arg2], ...);`

- Primer:

```
function mul(a, b) {
  return a * b;
}

let double = mul.bind(null, 2);  /* null je umesto context (this) koga ovde nemamo; ovim pozivom se pravi i vraća funkcija mul1(b) { return 2 * b}  */

alert( double(3) ); // = mul1(3) = 6
alert( double(4) ); // = mul1(4) = 8
alert( double(5) ); // = mul1(5) = 10
```

Metoda `bind` i parcijalne funkcije: još jedan primer

```
function mul1(a, b) {
  return a * b;
}
function mul2(a, b, c) {
  return a * b *c;
}
let double1 = mul1.bind(null, 2);  // null je umesto context
                                   // koga ovde nemamo
alert( double1(3) ); // = mul1(2, 3) = 6
alert( double1(4) ); // = mul1(2, 4) = 8
alert( double1(5) ); // = mul1(2, 5) = 10
let double2 = mul2.bind(null, 2, 3);
alert( double2(3) ); // = mul2(2, 3, 3) = 18
alert( double2(4) ); // = mul2(2, 3, 4) = 24
alert( double2(5) ); // = mul2(2, 3, 5) = 30
```

Kurirana funkcija$_{1/2}$

- ***Kurirana funkcija*** je funkcija sa više argumenata koja argumente (višestruke) prihvata **jedan po jedan**, vraćajući svaki put funkciju koja prihvata sledeći, JEDAN argument, sve dok se ne proslede svi argumenti.
- Zašto nam je potrebno kuriranje:
  - Teorijska osnova (jedna od dve) funkcionalnog programiranja je λ-račun gde funkcija f prima jednu ulaznu vrednost (x) i vraća jedan izlaz (t):

$$x \rightarrow \boxed{f} \rightarrow t$$

  - Da bismo mogli da podržimo funkcije više promenljivih (na primer, $g(x, y)$), potrebno je da obrađujemo jedan po jedan argument

Kurirana funkcija 2/2

- Primer: funkcija sa 2 argumenta
  - Nekurirana

```
// add = (a, b) => a + b => Number
const add = (a, b) => a + b;
const result = add(2, 3); // => 5
```

  - Kurirana

```
// add = a => b => Number
const add = a => b => a + b;
const result = add(2)(3); // => 5
```

Matematika: kururanje

- Matematika poznaje funkcije koje zavise od više argumenata:

$$f(x, y, z) = 256x + 48y - 23z$$

- Kako se sračunava vrednost funkcije za $x = 1, y = 2, z = 3$?
  - Može ovako:

$$f(1, 2, 3) = 256 * 1 + 48 * 2 - 23 * 3$$

  - A može i ovako (kuriranje):

$$f(1, y, z) \Rightarrow g(y, z) = 256 * 1 + 48y - 23z = 256 + 48y - 23z$$

$$g(2, z) \Rightarrow h(z) = 256 + 96 - 23z = 352 - 23z$$

$$h(3) \Rightarrow 352 - 23 * 3 = 352 - 69 = 283$$

- Mi, u stvari, u funkciji zamenjujemo “promenljive” njihovim vrednostima.
  - U prvom slučaju “odjednom”.
  - U drugom slučaju “jednu-po-jednu” pri čemu u svakom koraku primenjujemo različite funkcije koje nastaju kao rezultat svake zamene. Na kraju nema ništa da se zameni i funkcija je sračunata.

Kuriranje

- Tehnika slična parcijalnoj aplikaciji, gde se funkcija koja očekuje više argumenata razlaže na **sukcesivne ulančane funkcije arnosti 1** od kojih svaka prima po jedan argument i **vraća drugu funkciju** da prihvati sledeći, **opet samo jedan** argument.
- Kuriranje je u vezi sa parcijalnom aplikacijom utoliko što svaki sukcesivni kurirani poziv parcijalno primenjuje po jedan argument originalne funkcije dok se ne proslede svi argumenti.
- Ako je originalna funkcija očekivala pet argumenata, kurirana forma te funkcije bi preuzela samo prvi argument i vratila funkciju koja prihvata drugi argument. Ta funkcija bi primila samo drugi argument i vratila funkciju koja preuzima treći argument. Vraćena funkcija bi primila treći argument i vratila funkciju koja prihvata četvrti argument. Ova funkcija bi prihvatila četvrti argument i vratila funkciju koja prihvata peti argument. Vraćena funkcija je poslednja funkcija koja na kraju prihvata peti argument i izračunava vrednost polazne funkcije.

Funkcija curry()

```
function curry(fn, arity = fn.length) {
   return (function nextCurried(prevArgs){
   return function curried(nextArg){
   var args = [ ...prevArgs, nextArg ];

   if (args.length >= arity) {
   console.log('prevArgs na kraju:', prevArgs)
   console.log('nextArg na kraju:', nextArg)
   console.log('args na kraju:', args)
      return fn( ...args ); // izvršavanje fn
                            }
   else {
   console.log('prevArgs u toku:', prevArgs)
   console.log('nextArg u toku:', nextArg)
   console.log('args u toku:', args)
 // ono što je prikupljeno u args je prevArgs za nextCurried()
      return nextCurried( args );
        }
        };
    })( [] );
}
```

Pozivanje funkcije **curry()** 1/2

```
function foo(x,y,z) {// funkcija koja se kurira
    let result = 256*x+48*y+23*z
  return result
  }

console.log (curry(foo)(1)(2)(3)) // 421
```

Pozivanje funkcije `curry()` 2/2

- prevArgs : [ ] - počinje se sa praznim nizom prevArgs jer ni jedan argument još nije prosleđen
- **nextArg : 1** - uzima se prvi argument
- **args: [1]** - uzeti argument se dodaje u niz args.
- **prevArgs: [1]** – u prevArgs je sada 1 – to je prosleđeni argument
- **nextArg: 2** – uzima se drugi argument
- **args: [1, 2]** - uzeti argument se dodaje u niz args.
- **prevArgs : [1, 2]** – u prevArgs su sada 1 i 2 – to su prosleđeni argumenti
- **nextArg: 3** - uzima se treći (posledni) argument
- **args na kraju: [1, 2, 3]** – u nizu args su svi argumenti
- Poziva se funkcija foo (args) i ona izračunava vrednost 421

Kuriranje: svojstvo `length`

- Za određivanje broja kuriranja potrebnih za procesiranje svih argumenata funkcije naša implementacija koristi vrednost svojstva `length` funkcije koja se kurira.
- Ima i situacija u kojima svojstvo `length` ne sadrži odgovarajuću vrednost za korektno kuriranje pa o tome treba voditi računa :
  - Ako parametarska signatura originalne funkcije uključuje podrazumevane vrednosti parametra. U tom slučaju vrednost svojstva `length` je umanjena za broj parametara sa podrazumevanom vrednošću.
  - Ako parametarska signatura originalne funkcije uključuje destrukturiranje parametra (`rest` sintaksa). U tom slučaju, nizovni parametar se računa kao **jedan** parametar u svojstvu `length`.
  - Ako parametarska signatura originalne funkcije opisuje varijadičku funkciju (funkcija sa promenljivim brojem parametara). U tom slučaju JavaScript postavlja svojstvo `length` na vrednost 0.

Kuriranje i parcijalna aplikacija

- Pri parcijalnoj aplikaciji, funkcija (***n***-arna) može da prihvati **proizvoljan broj** (***k***) svojih argumenata u jednom trenutku. To se radi tako što funkcija kao rezultat vraća drugu ***m***-arnu funkciju gde je ***m***=***n***-***k***.
- Pri kuriranju, funkcija *uvek* ***prima jedan argument*** i **vraća unarnu funkciju** - funkciju koja prihvata ***jedan argument***.
- Zahtev **unarnosti** je svojstvo kuriranih funkcija koji nije nametnut parcijalnoj aplikaciji.
- Sve kurirane funkcije vraćaju parcijalne aplikacije, ali nisu sve parcijalne aplikacije kurirane funkcija.

Point-free stil$_{1/2}$

- **Point-free stil** je stil programiranja gde **definicije funkcija ne referenciraju argumente funkcije**.
- Defincije funkcija u JS-u:

```
function foo (/*ovde se deklarišu parametri*/) {
// ...
}

const foo = (/*ovde se deklarišu parametri*/) => // ...

const foo = function (/* ovde se deklarišu parametri*/) {
// ...
}
```

- Kako definisati funkciju u JS-u bez deklarisanja zahtevanih parametara? Naravno, koristeći ***zatvaranje***

Point-free stil 2/2

- Zadatak: Definisati funkciju u JS-u bez deklarisanja zahtevanih parametara.
- Rešenje: Pozivanjem funkcije koja vraća funkciju.
- Primer: U point-free stilu kreirati funkciju `inc()` koja svaki broj koji joj se prosledi uvećava za 1. Pri tome koristiti funkciju `add(a,b)`.

Point-free `inc()` 1/2

- Kurirana funkcija `add` prihvata broj i vraća parcijalno primenjenu funkciju sa prvim parametrom fiksiranim na bilo šta što se prosledi.

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

- Dakle: add(1) je **naša funkcija**
- Možemo da je iskoristimo da kreriamo novu funkciju `inc()`:

```
// inc = n => Number
// Dodaje 1 na bilo koji broj.
const inc = add(1);
inc(3); // => 4
```

Point-free `inc()` 2/2

- Ovo je mehanizam zgodan za generalizaciju i specijalizaciju.
- Vraćena funkcija je samo *specijalizovana verzija* opštije funkcije kao što je `add()`.
- Koristeći funkciju `add()` možemo kreirati koliko želimo specijalizovanih verzija:

```
const inc10 = add(10);
const inc20 = add(20);
inc10(3); // => 13
inc20(3); // => 23
```

Point-free `inc()`: kako radi

- Kada se kreira `inc()` funkcijskim pozivom `add(1)`, parametar `a` funkcije `add()` se *fiksira* na 1 unutar vraćene funkcije (`inc()`) koja prima jedan argument.
- Kada se, zatim, pozove `inc(3)`, on ima zapamćenu vrednost `a` = 1 a prosleđena vrednost 3 se dodaje na zapamćeno `a` i aplikacija se kompletira vraćajući 1 + 3 = 3.
- Sve kurirane funkcije su oblik funkcija višeg reda koji omogućuje kreiranje verzija originalne funkcije specijalizovanih za određenu namenu.

Ko o čemu, a mi o zatvaranju

- Dakle, zatvaranje je od suštinskog značaja i u point-free notaciji.
- Najjednostavnije rečeno, **zatvaranje** je ***unutrašnja funkcija***.
- A šta je unutrašnja funkcija? Pa, to je ***funkcija unutar druge funkcije***. Nešto kao:

```
function outer() {
    function inner() {
    }
}
```

- Razlog što je zatvaranje tako moćan mehanizam je što je ono u stanju da pristupa *lancima dosega* (zovu se još i *nivoi dosega*).

Da se podsetimo: čemu zatvaranje može da pristupi?

- Tehnički, zatvaranje ima pristup do tri dosega:
  - Svom lokalnom dosegu - sopstvenim varijablama koje su deklarisane u deklaracijama koje se pojavljuju u samoj funkciji koja vrši zatvaranje.
  - Globalnom dosegu - varijablama koje su deklarisane u globalnom dosegu.
  - Lokalnom dosegu roditeljske funkcije (funkcija u kojoj je funkcija zatvaranja definisana) - varijablama deklarisanim u roditeljskoj funkciji i nakon izvršenja roditeljske funkcije (**ovo je posebno značajno!**).

Zatvaranje: pristup sopstvenim i tuđim varijablama

```
let global = "global"
  function spoljna() {
  let roditeljska = "roditeljska"
     function unutrasnja() {
       let a = 5;
         console.log(' a: ' + a)
          console.log(' roditeljska: ' + roditeljska)
         console.log(' global: ' + global)
     }
  unutrasnja() //poziv funkcije unutrasnja.
}
spoljna();
```

- Svi ispisi se izvršavaju u funkciji **`unutrasnja():`**
  - **`a :5`** (a je sopstvena varijabla funkcije **`unutrasnja()`**)
  - `roditeljska:` **`roditeljska`** (`roditeljska` je tuđa varijabla iz lokalnog dosega funkcije **`spoljna()`**)
  - `global:` **`global`** (`global` je tuđa varijabla iz globalnog dosega)

Zatvaranje koje pamti svoj kontekst

```
var fn = (arg) => {
    let spoljna = "Vidljivo"
    let unFn = () => {
      console.log(spoljna)
      console.log(arg)
   }
   return unFn
}

let pet = 5
let zatvaranjeFn = fn(pet);
    zatvaranjeFn();
```

- Rezultat je ispis:

Vidljivo
5

Još o zatvaranju: Funkcija `tap`

```
const tap = (value) =>
  fn => (
    typeof(fn) === 'function' && fn(value),
    console.log(value)
)

tap("Ulaz u Tap")(it => console.log("U fn
 argument value je ",it))
```

Kompozicija funkcija

- Šta je kompozicija funkcija
- Redosled izvršavanja i redosled navođenja
- Premeštanje parametara/argumenata funkcije
- Opšta kompozicija funkcija

Šta je kompozicija funkcija

- U računarskoj nauci, kompozicija funkcija je čin ili mehanizam kojim se jednostavne funkcije(komponente) kombinuju da bi se napravile složenije funkcije (kompoziti).
- Kao i kod uobičajene kompozicije funkcija u matematici, mehanizam (pa i sam akt) komponovanja funkcija svodi se na prenošenje rezultata svake komponentne funkcije sledećoj funkciji putem argumenta, a rezultat poslednje funkcije je konačan rezultat kompozitne funkcije.

Kompozicija funkcija: primeri

- U matematici

$$f(x) = \sin(e^x) = g\big(h(x)\big); h(x) = e^x; g(x) = \sin(x)$$

- U kodu

```
function compose2(fn2, fn1) {
   return function
     composed(origVrednost){
       return fn2( fn1(origVrednost) );
       };
   }
   function f1(x){return 2*x}
   console.log(compose2(f1,f1)(1)) // => 4
```

Kompozicija: redosled izvršavanja funkcija

- Funkcije se komponuju sa desna na levo. To je tako i u matematici:
  `f(g(x))` – prvo se izvršava **`g()`**, zatim **`f()`**.
- A to je standard i u JS bibliotekama – prvo se izvršava funkcija **`fn1`** a zatim **`fn2,`** a u listi parametara se navode obrnutim redosledom:

```
function compose2(fn2, fn1) {
return function
  composed(origVrednost){
    return fn2( fn1(origVrednost) );
    };
}
```

Kompozicija: redosled izvršavanja funkcija

```
function compose2(fn2, fn1) {
return function
  composed(origVrednost){
    return fn2( fn1(origVrednost) );
    };
}
function f1(x){return 2*x}
console.log(compose2(f1,f1)(1)) // => 4
```

Premeštanje parametara/argumenata

- I jezik matematike i jezik programiranja nameću ograničenja na redosled parametara/argumenata pri radu sa funkcijama.
- Recimo da imamo funkciju **`func(a, b, c)`**. Šta može da se uradi ako poželimo da prvo parcijalno primenimo **c** a da sačekamo da kasnije specificiramo **a** i **b**?
- Naravno, rešenje je funkcija koja obmotava funkciju `func()` i to tako da preokrene redosled njenih argumenata. Sledi takva funkcija.

Funkcija koja obrće redosled argumenata

```
function reverseArgs(fn) {
    return function argsReversed(...args){
    return fn( ...args.reverse() );
        };
}

function func (a,b,c){
   return a+b-c}
console.log(func(1,2,3)) // 1+2-3 = 0
const funcReversed = reverseArgs(func)
console.log(funcReversed(1,2,3)) // 3+2-1 = 4
```

Restaurisanje redosleda

- Da bi se restaurisao redosled, treba preokrenuti redosled argumenata uzastopno parcijalno primenjenih funkcija. Dakle, pozove se `reverseArgs()` nad funkcijom sa invertovanim parametrima:

```
function func (a,b,c){
    return a+b-c}

console.log(func (1,2,3)) // => 0
const funcReversed = reverseArgs(func)
console.log(funcReversed(1,2,3)) // => 4
const funcRestored = reverseArgs(funcReversed)
console.log(funcRestored(1,2,3)) // => 0
```

Funkcija `partialRight()`

```
function partialRight(fn,...presetArgs) {
    return reverseArgs(
    partial( reverseArgs(fn), ...presetArgs.reverse()
  )
    );
}

// Korišćenje – primeni parcijalno sa desna
var cacheResult = partialRight(ajax, function
  onResult(obj){
    cache[obj.id] = obj;
});

// Kasnije – radi normalno
cacheResult( "http://some.api/person", {
  user:CURRENT_USER_ID } );
```

Performantnija funkcija **partialRight()**

```
function partialRight(fn,...presetArgs) {
  return function
   partiallyApplied(...laterArgs){
    return fn(...laterArgs, ...presetArgs);
        };
}
```

Šta se ne garantuje/garantuje

- Ni jedna od ovih implementacija funkcije `partialRight()` **ne garantuje** da će **specifični parametar primiti specifičnu parcijalno primenjenu vrednost**;
- One **samo garantuju** da će **parcijalno primenjena(e) vrednost(i) da se pojavi(e) na krajnjoj desnoj (poslednjoj) poziciji** u listi argumenata prosleđenih originalnoj funkciji.

Primer inverzije parametara

```
function foo(x,y,z,...rest) {
    console.log( x, y, z, rest );
}

var f = partialRight( foo, "z:last" );

f( 1, 2 );    // Vraća: 1 2 "z:last" []
f( 1 );      // Vraća: 1 "z:last" undefined []
f( 1, 2, 3 );// Vraća: 1 2 3 ["z:last"]
f( 1, 2, 3, 4 );// Vraća: 1 2 3 [4,"z:last"]
```

Opšta kompozicija funkcija

- Primer kompozicije koji smo do sada videli komponuje dve funkcine.
- Opšta kompozicija funkcija znači da možemo da komponujemo proizvoljan (konačan) broj funkcija

```
finalValue <-- f1 <-- f2 <-- ... <-- fN <-- origValue
```

- A **pomoću čega** se pravi kompozicija?
  - Naravno, pomoću **funkcije** `compose()`

Funkcija compose()

```
function compose(...fns) {
return function composed(result){
// kopiranje niza funkcija
var list = [...fns]; // zbog toga što .pop() mutira
                     // listu u funkciji composed()

while (list.length > 0) {

/* Zato što se funkcije izvršavaju sa desna na levo,
uzmi funkciju sa kraja liste (list.pop) i izvrši je
Napomena: operacija Array.pop() vraća element sa kraja liste
i uklanja ga iz liste */

       result = list.pop()( result );
      }

return result;
    };
}
```

Funkcija `compose()`: korišćenje

```
function foo0(x) {
    return 256*x
 }
function foo1(x) {
    return -x
 }
function foo2(x) {
    return -x
 }
let composedFunction = compose(foo0,foo1,foo2)
console.log(composedFunction(1))// => 256
```

Zadatak za ilustraciju kompozicije funkcija

- Dat je string u kome je tekstualni sadržaj takav da su reči razdvojene znakom blanko ili znakom \. Napisati program u jeziku JavaScript koji će iz zadatog teksta formirati niz koji sadrži pojedinačne reči iz teksta napisane malim slovima koje su duže od zadatog broja karaktera i to tako da nema ponavljanja istih reči u izlaznom nizu.

Rešenje bez funkcija

```
var text ="Da bi se komponovale dve funkcije, treba proslediti \
izlaz poziva prve funkcije \
na ulaz pozivu druge funkcije.";

let napravljeneReci =
   text
      .toLowerCase()
      .split( /\s|\b/ );
console.log (napravljeneReci)

var duzeReci = [];
let minimalnaDuzinaReci = prompt("Unesite minimalnu dužinu reči:");

for (let word of napravljeneReci) {
if (word.length > minimalnaDuzinaReci)
            duzeReci.push( word );
        }
 console.log (duzeReci)

var jedinstvenaLista = [];
   for (let v of duzeReci) {
   // vrednost još nije u listi?
      if (jedinstvenaLista.indexOf( v ) === -1 )
         jedinstvenaLista.push( v );
      }
   console.log (jedinstvenaLista)
```

Rešenje sa kompozicijom više funkcija 1/4

```
var text "Da bi se komponovale dve funkcije,
  treba proslediti \
izlaz poziva prve funkcije \
na ulaz pozivu druge funkcije.";

var duzeReci = compose( preskociKraceReci,
  izbaciDuplikate, napraviReci);
var korisceneReci = duzeReci( text );
console.log (korisceneReci);

/* ['komponovale', 'funkcije', 'treba',
  'proslediti', 'izlaz', 'poziva', 'pozivu',
  'druge'] */
```

Primer kompozicije više funkcija 2/4

```
function preskociKraceReci (words) {
var filteredWords = [];
var minWordLength = prompt("Unesite minimalnu
  dužinu reči:");

for (let word of words) {
if (word.length > minWordLength) {
            filteredWords.push( word );
        }
    }

return filteredWords;
}
```

Primer kompozicije više funkcija 3/4

```
function napraviReci(str) {
    return String( str )
        .toLowerCase()
        .split( /\s|\b/ )
        .filter( function alpha(v){
            return /^[\w]+$/.test( v );
    } );
}
```

Primer kompozicije više funkcija 4/4

```
function izbaciDuplikate(list) {
    var uniqList = [];
    for (let v of list) {
    // value not yet in the new list?
        if (uniqList.indexOf( v ) === -1 ) {
            uniqList.push( v );
        }
    }
    return uniqList;
}
```

Predefinisanje “kostura” kompozicije

- Desna parcijalna aplikacija se uradi da se fiksira “kostur” kompozicije (putem nekih argumenata)
  - U primeru: Uradi se desna parcijalna aplikacija `compose()` tako što će se fiksirati drugi i treći argument (`izbaciDuplikate()` i `napraviReci()`, respektivno); ono što se dobije naziva se, recimo, `filterWords()`.
- Zatim se kompozicija može kompletirati višestrukim pozivanjem funkcije `filterWords()`, ali sa različitim prvim argumentima koji bliže određuju ponašanje (duge reči/kartke reči, sa duplikatima/bez duplikata)
- To znači da se "kostur" kompozicije može unapred definisati i da, nakon toga, mogu da se kreiraju specifične varijacije te kompozicije.
  - U ovom primeru "kostur" je izdvajanje reči, a specifične varijacije su filtriranje dužih ili kraćih reči, po želji.
- Ovaj obrazac je je jedan od najmoćnijih obrazaca u funkcionalnom programiranju!

Predefinisanje “kostura” kompozicije

```
function preskociDugeReci(list) { /* Ovde dođe kod */
  }
// Kostur kompozicije
var filterWords  = partialRight(compose,
  izbaciDuplikate, napraviReci);
// Varijacije kompozicije
var duzeReci  = filterWords(preskociKratkeReci);
var kracReci  = filterWords(preskociDugeReci);

duzeReci( text );
// ["compose","functions","together","output","first",
// "function","input","second"]

kraceReci( text );
// ["to","two","pass","the","of","call","as"]
```

Kompozicija `pipe()`

- Postoji i komponovanje inverznim redosledom sa leva na desno koje ima zajedničko ime - zove se: `pipe()`.
- Smatra se da ovo ime dolazi iz Unix/Linux okruženja, gde se višestruki programi nižu zajedno putem "pipe“-ing-a (| operator) - izlaz prvoga je ulaz u drugi, itd. (n.pr.,

```
ls -la | grep "foo" | less
```

- `pipe()` je isti kao i `compose()`, izuzev što kroz listu funkcija prolazi po uređenju sa leva-na-desno

Funkcija `pipe()`

```
function pipe(...fns) {
return function piped(result){
var list = [...fns];

while (list.length > 0) {
// uzmi prvu funkciju iz liste i izvrši je
            result = list.shift()(result);
        }

return result;
    };
}
```

U stvari:

```
var pipe = reverseArgs(compose);
```

Odnosno:

```
var biggerWords = compose(skipShortWords,
 unique, words);
var biggerWords = pipe(words, unique,
 skipShortWords);
```

Prednosti `pipe()` u odnosu na `compose()`

- `pipe()` je zgodan i ako treba parcijalno primeniti *prvu(e)* funkciju(e) koja(e) se izvršava(ju).
- Prethodno je to rađeno desnom parcijalnom aplikacijom funkcije `compose()`:

```
var filterWords = partialRight(
  compose, unique, words );
```

- A pomoću `pipe()` je:

```
var filterWords = partial( pipe, words,
  unique );
```

Sažetak

- **Funkcija višeg reda** je funkcija koja: može da primi drugu funkciju kao argument, može da vrati drugu funkciju kao izlaz, može i jedno i drugo (da primi funkciju kao argument i da vrati funkciju)
- **Parcijalna aplikacija** (parcijalna aplikacija funkcije) je obrazac fiksiranja određenog broja argumenata funkcije koji za rezultat ima funkciju arnosti koja je jednaka arnosti originalne funkcije umanjenoj za broj fiksiranih argumenata.
- **Kuriranje** je obrazac fiksiranja argumenta funkcije kojim se kao rezultat vraća unarna funkcija.
- **Point-free stil** je stil programiranja gde definicije funkcija ne referenciraju argumente funkcije. Ostvaruje se korišćenjem **zatvaranja**.
- **Kompozicija funkcija** je čin ili mehanizam kojim se jednostavne funkcije(komponente) kombinuju da bi se napravile složenije funkcije (kompoziti).
  - Opšta kompozicija komponuje funkcije sa desna na levo.
  - Pipe kompozicija komponuje funkcije sa leva na dasno.

Literatura za predavanje

1. Z. Konjović, Funkcionalno programiranje, **Tema 03 - Funkcije višeg reda**, slajdovi sa predavanja, dostupni na folderu **FP kurs 2023-24→Files →Slajdovi sa predavanja**
2. Eric Elliot, **Composing Software - An Exploration of Functional Programming and Object Composition in JavaScript**, Leanpub, 2019, Poglavlje "**Higher Order Functions**"
3. Kyle Simpson, **Functional-Light JavaScript - Balanced, Pragmatic FP in JavaScript**, Manning, 2018, Poglavlje "The Nature Of Functions "