Univerzitet Singidunum
www.singidunum.ac.rs

Funkcionalno programiranje

Školska 2023/24 godina

Letnji semestar

Tema 5: Rekurzija

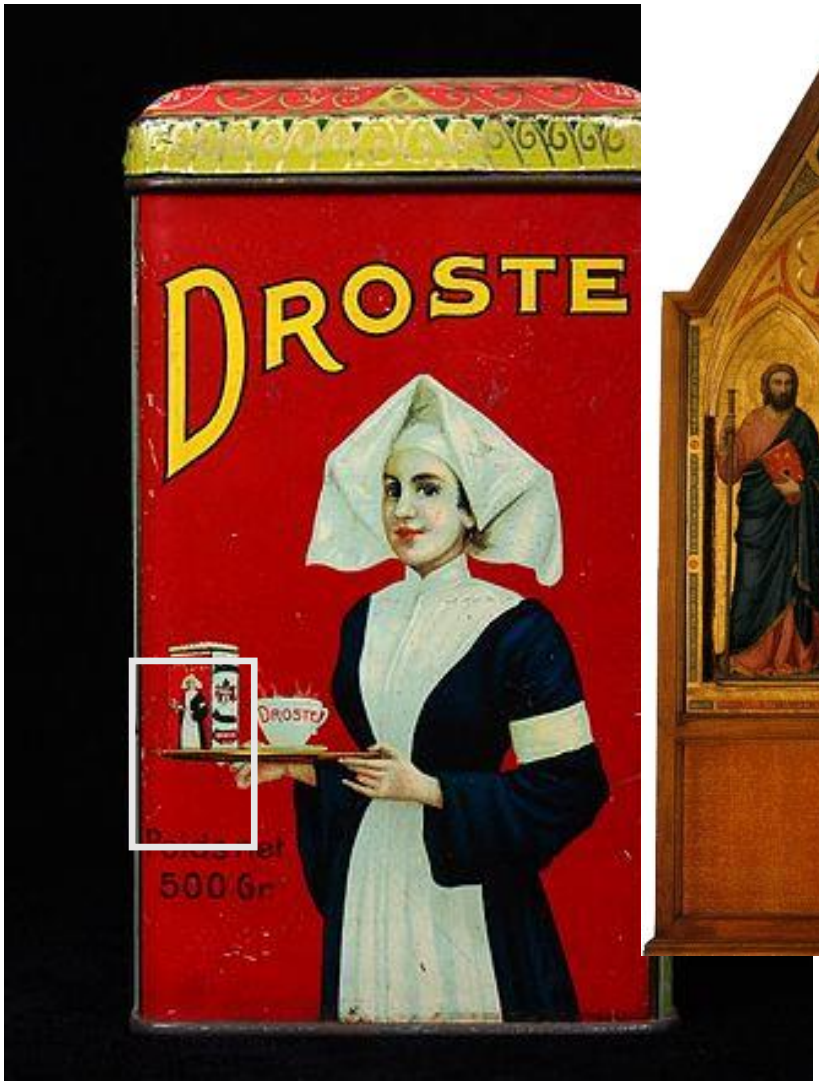
Sadržaj

- Šta je rekurzija
- Rekurzija i stek
  - Preuređivanje rekurzije
- Zašto rekurzija
- Deklarativnost rekurzije
- Rekurzija i imutabilnost
- Memoizacija

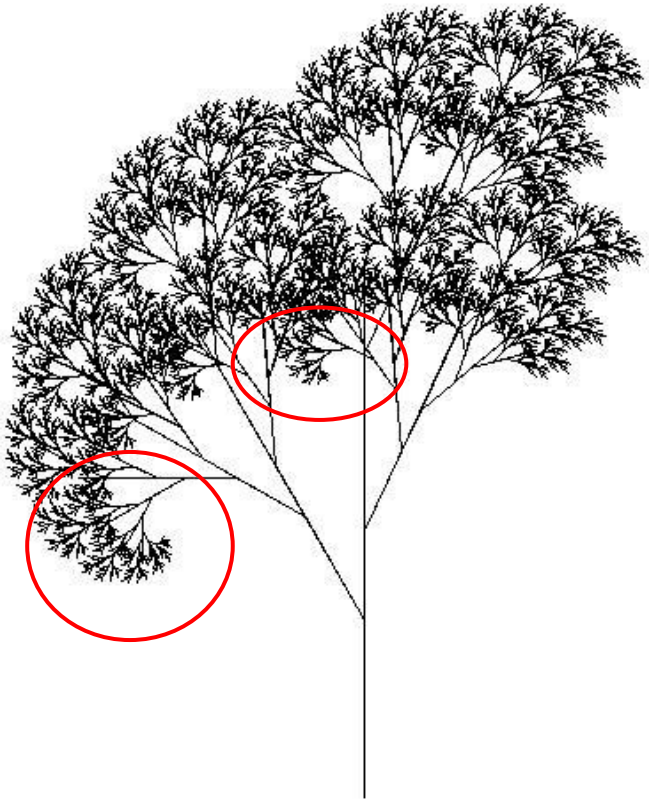
Definicija rekurzije

- Lingvistika: ***Rekurzija*** je
  - (1) "ponovljeno sekvencijalno korišćenje određenog tipa lingvističkog elementa ili gramatičke strukture. ",
  - (2) "procedura koja samu sebe poziva, ili . . . konstituenta koja sadrži konstituentu iste vrste. “
- Matematika i računarska nauka:
  - Klasa objekata ili metoda ispoljava rekurzivno ponašanje ako se može definisati putem dva svojstva:
    - Jednostavan ***bazni slučaj*** (ili ***slučajevi***) - terminirajući scenario koji ne koristi rekurziju za kreiranje odgovora.
    - ***Rekurzivni korak*** - skup pravila koji sve ostale slučajeve svodi na bazni slučaj.

Rekurzija u umetnosti



Rekurzija u “digitalnoj umetnosti”



Značaj rekurzije u programiranju

- “Moć 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.”

Niklaus Virt, *Algoritmi + strukture podataka = Programi*, 1976

Rekurzija: Bazni slučajevi i rekurzivni koraci

- Definicija rekurzivne funkcije ima jedan ili više osnovnih slučajeva, što znači ulaz(e) za koje funkcija proizvodi rezultat trivijalno (bez ponavljanja), i jedan ili više rekurzivnih slučajeva, što znači ulaz(e) za koje se program ponavlja (poziva sebe) .
- Primeri
  - Nečiji predak (je):
    - Nečiji roditelj — bazni slučaj
    - Predak nečijeg roditelja — rekurzivni korak
  - Fibonačijev niz

    $Fib(0) = 0$ — bazni slučaj 1

    $Fib(1) = 1$ — bazni slučaj 2

    Rekurzivni korak:

    Za svako celo $n > 1, Fib(n) = Fib(n-1) + Fib(n-2)$

Osnovne vrste rekurzije: pojedinačna rekurzija

- Rekurzija koja u rekurzivnom koraku sadrži samo jednu samoreferencu zove se **pojedinačna (jednostruka) rekurzija**
- Tipičan primer jednostruke rekurzije je računanje faktorijela:

```
// f(n) = n * (n-1) * (n-2) ... * 3 * 2 * 1
function factorial (n){
// bazni korak
   if (n == 0){
      return 1
   }
// rekurzivni korak
   else {
      return n*factorial(n-1)
   }
}
factorial(5) // => 120
```

Osnovne vrste rekurzije: višestruka rekurzija

- Rekurzija koja sadrži više samoreferenci u rekurzivnom koraku zove se **višestruka rekurzija**.
- Na primer:

Fibonačijev niz

$Fib(0) = 0$ bazni slučaj 1

$Fib(1) = 1$ bazni slučaj 2

Rekurzivni korak:

Za svako celo $n > 1, Fib(n)$= $\boldsymbol{Fib(n-1) + Fib(n-2)}$

Osnovne vrste rekurzije: direktna rekurzija

- Kada funkcija poziva samu sebe, to se zove ***direktna rekurzija***.
- Na primer:

```
function foo(x) {
    if (x < 5) return x; //bazni korak
        return foo(x/2); // vraća poziv funkcije same
                         // sebe - rekurzivni korak
}
foo(16); // => 4
```

Osnovne vrste rekurzije: uzajamna rekurzija

- Kada **dve funkcije pozivaju jedna drugu** to se zove **uzajamna (cirkularna) rekurzija** (funkcija f_1 pozove funkciju f_2 pa funkcija f_2 pozove funkciju f_1). Na primer:

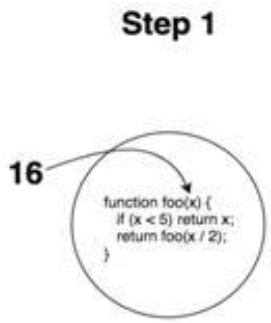
```
function je_paran(n) {
    if (n == 0)
        return true
    else
        return je_neparan(n - 1)
}

function je_neparan(n) {
    if (n == 0)
        return false
    else
        return je_paran(n - 1)
}
console.log(je_paran(5)) //=> false
console.log(je_paran(4)) //=> true
```

Osnovne vrste rekurzije: uzajamna rekurzija

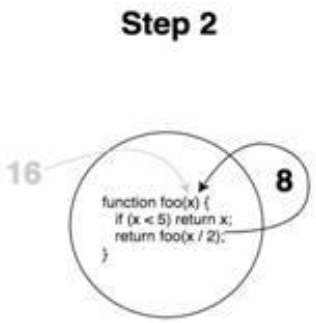
- Kada **dve funkcije** (ili više funkcija) **pozivaju jedna drugu** u rekurzivnom ciklusu, to se zove ***indirektna (uzajamna) rekurzija***.
- Na primer: funkcija f_1 poziva funkciju f_2, funkcija f_2 poziva funkciju f_3, … f_{n-1} poziva f_n a onda f_n pozove f_{n-1}.

Rekurzija: malo koda, malo slike

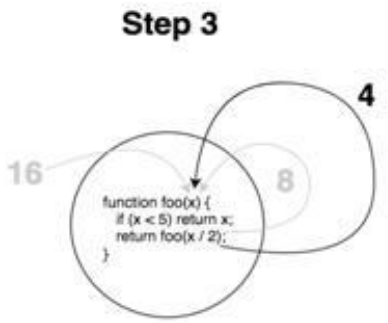
```
function foo(x) {
   if (x < 5) return x; //vraća vrednost x – bazni korak
      return foo(x/2); // vraća poziv funkcije foo sa x/2
                       // - rekurzivni korak
}
foo(16);
```



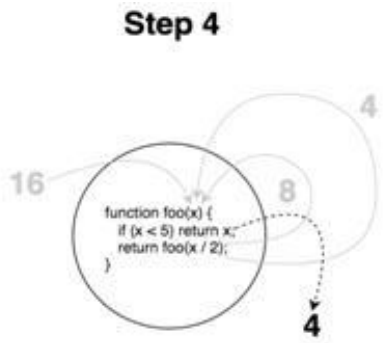
Rekurzivni poziv **foo(16)**



Rekurzivni poziv **foo(8)**

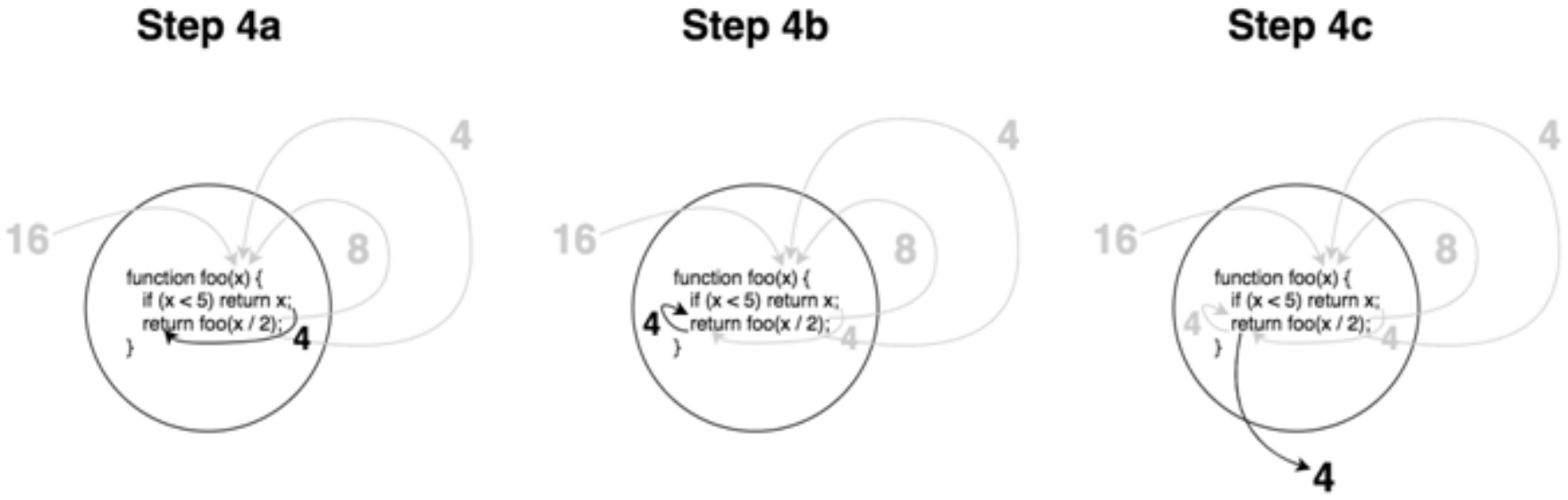


Rekurzivni poziv **foo(4)**

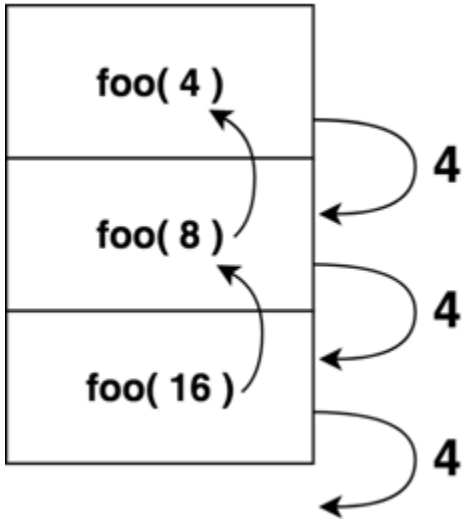


Nema poziva **foo()** – izvršava se **return x**

Rekurzija: nakon zadovoljenja baznog uslova



Rekurzija: Stek poziva



Rekurzija i stek poziva: primer faktorijel funkcije$_{1/3}$

```
function factorial (x){
// bazni uslov
    if (x == 0){
        return 1
    }
// rekurzivni korak
    else {
        return x*factorial()x-1
    }
```

Rekurzija i stek poziva: primer faktorijel funkcije$_{2/3}$

```
factorial(5)
├─ factorial(4)
│  ├─ factorial(3)
│  │  ├─ factorial(2)
│  │  │  ├─ factorial(1)
│  │  │  │  └─ factorial(0) => vraća 1
│  │  │  └─ return 2 * factorial(1) => vraća 2
│  │  └─ return 3 * factorial(2) => vraća 6
│  └─ return 4 * factorial(3) => vraća 24
└─ return 5 * factorial(4) => vraća 120
```

Rekurzija i stek poziva: primer faktorijel funkcije$_{3/3}$

- Da bi se vratila sračunata vrednost, JavaScript endžin mora da vrati svaki poziv funkcije `factorial()` redosledom koji je obrnut od redosleda pozivanja:

- Šesti poziv: `factorial(0) //vraća 1`
- Peti poziv: `factorial(1) //vraća 1 * factorial(0) = 1*1 = 1`
- Četvrti poziv: `factorial(2) // vraća 2 * factorial(1) = 2*1 = 2`
- Treći poziv: `factorial(3) // vraća 3 * factorial(2) = 3*2 = 6`
- Drugi poziv: `factorial(4) // vraća  4 * factorial(3) = 4*6 = 24`
- Prvi poziv: `factorial(5) // vraća  5 * factorial(4) = 5*24 = 120`

Rekurzija: Fibonačijev niz - direktna rekurzija

```
function fib(n) {
    if (n <= 1) return n;
        return fib(n - 2) + fib(n - 1);
}
```

Rekurzija: Fibonačijev niz – uzajamna rekurzija

```
function fib_(n) {
    if (n == 1) return 1;
    else
        return fib(n - 2);
}

function fib(n) {
    if (n == 0) return 0;
    else
        return fib(n - 1) + fib_(n);
}
```

Zašto rekurzija

- Najčešće obrazloženje korišćenja rekurzije je da se ona uklapa u duh funkcionalnog programiranja, jer balansira (dobar deo) eksplicitnog praćenja stanja sa implicitnim stanjem na steku poziva.
  - Rekurzija je najkorisnija u problemima koji zahtevaju uslovna grananja i bek-treking, jer upravljanje takvim vrstama stanja u čisto iterativnom okruženju može da bude dosta komplikovano; u najmanju ruku, kod je visoko imperativan i teži za čitanje i verifikaciju.
- Praćenje svakog nivoa grananja kao sopstvenog dosega na steku poziva značajno olakšava čitanje koda.

Rekurzivni i iterativni algoritmi: trivijalan primer

```
// imperativno
function sum(total,...nums) {
   for (let num of nums) {
        total = total + num;
   }

   return total;
}

// deklarativno
function sum(num1,...nums) {
   if (nums.length == 0) return num1;
      return num1 + sum(...nums);
}
```

Deklarativnost rekurzije

- U matematici, notacija se koristi da se postigne čitljivost, na primer
  - $\sum x_i$ je mnogo čitljivije od $x_1 + x_2 + \ldots + x_n$
- Rekurzija je deklarativna za algoritme na isti način na koji je simbol Σ deklarativan za matematiku.
- Rekurzija izražava da rešenje problema postoji, ali ne zahteva od čitaoca koda da u tančine razume kako to rešenje radi.

Deklarativnost rekurzije: Primer

- Zadatak: Nalaženje najvećeg parnog broja prosleđenog kao argument
- Pokazaćemo dva pristupa implementaciji:
  - Imperativno, i
  - Deklarativno

Nalaženje najvećeg parnog: imperativna petlja

```
function maxParan(...brojevi) {
var maxBroj = -Infinity;

for (let broj of brojevi) {
if (broj % 2 == 0 && broj > maxBroj) {
            maxBroj = broj;
        }
    }

if (maxBroj !== -Infinity) {
return maxBroj;
    }
}
```

Nalaženje najvećeg parnog: rekurzivno

Signatura

maxParan(...nums): maxParan(nums.0, maxParan(...nums.1))

- Na primer:

maxParan(1, 10, 3, 2):

maxParan(1, maxParan(10, maxParan(3, maxParan(2)))

- JS rekurzivna definicija funkcije maxEven():

```
function maxParan(broj1,...ostaliBrojevi) {
    var maxOstali = ostaliBrojevi.length > 0 ?
    maxParan(...ostaliBrojevi ) : undefined; // bazni uslov

        return (broj1 % 2 != 0 || broj1 < maxOstali) ?
          maxOstali : broj1;
}
```

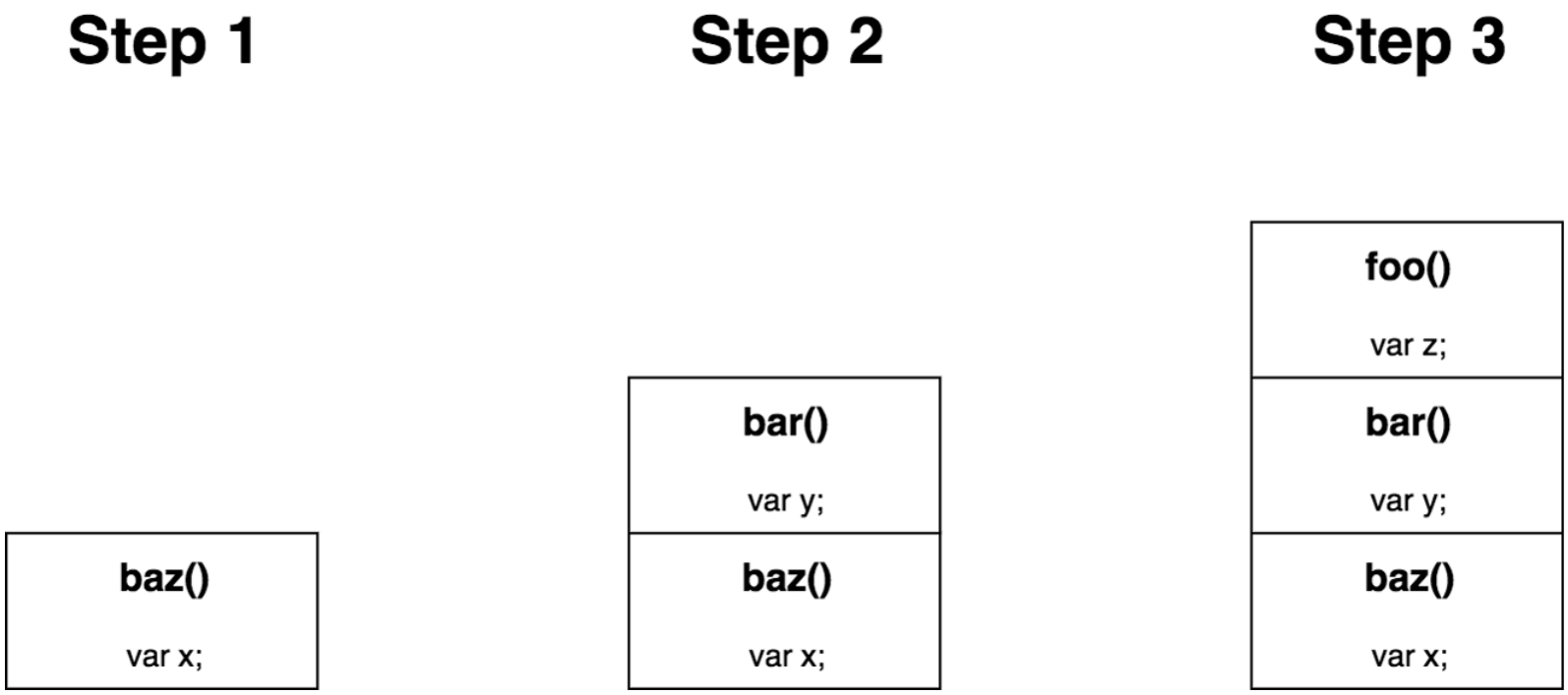
Rekurzija i stek

- Jedan od razloga za relativno skroman obim primene rekurzije njena zahtevnost za resursima, što se posebno odnosi na stek.
- Svaki funkcijski poziv uzima mali deo memorije koji se zove *aktivacioni zapis* (engl. *stack frame*).
  - Aktivacioni zapis sadrži potrebne informacije o tekućem stanju obrade naredbi u funkciji, uključujući i vrednosti svih varijabli.
  - Aktivacioni zapis mora da postoji sve dok se ne završi izvršavanje odgovarajućeg poziva funkcije.
- Funkcija može da pozove drugu funkciju što pauzira izvršenje tekuće funkcije.
- Kada se druga funkcija završi, endžin mora da nastavi izvršavanje prethodne funkcije a za to su mu potrebne informacije o stanju u kome je ta funkcija bila u trenutku pauziranja.Kada započne drugi funkcijski poziv, i njemu treba aktivacioni zapis pa se uzima još memorije i tako redom.
- Memorija je ograničena, pa može da dođe do prekoračenja steka, odnosno “ispadanja” iz memorije

Rekurzija i stek: ponašanje aktivacionog zapisa - kod

```
function foo() {
   var z = "foo!";
}

function bar() {
   var y = "bar!";
foo();
}

function baz() {
   var x = "baz!";
   bar();
}

baz();
```

Rekurzija i stek: ponašanje aktivacionog zapisa u slici



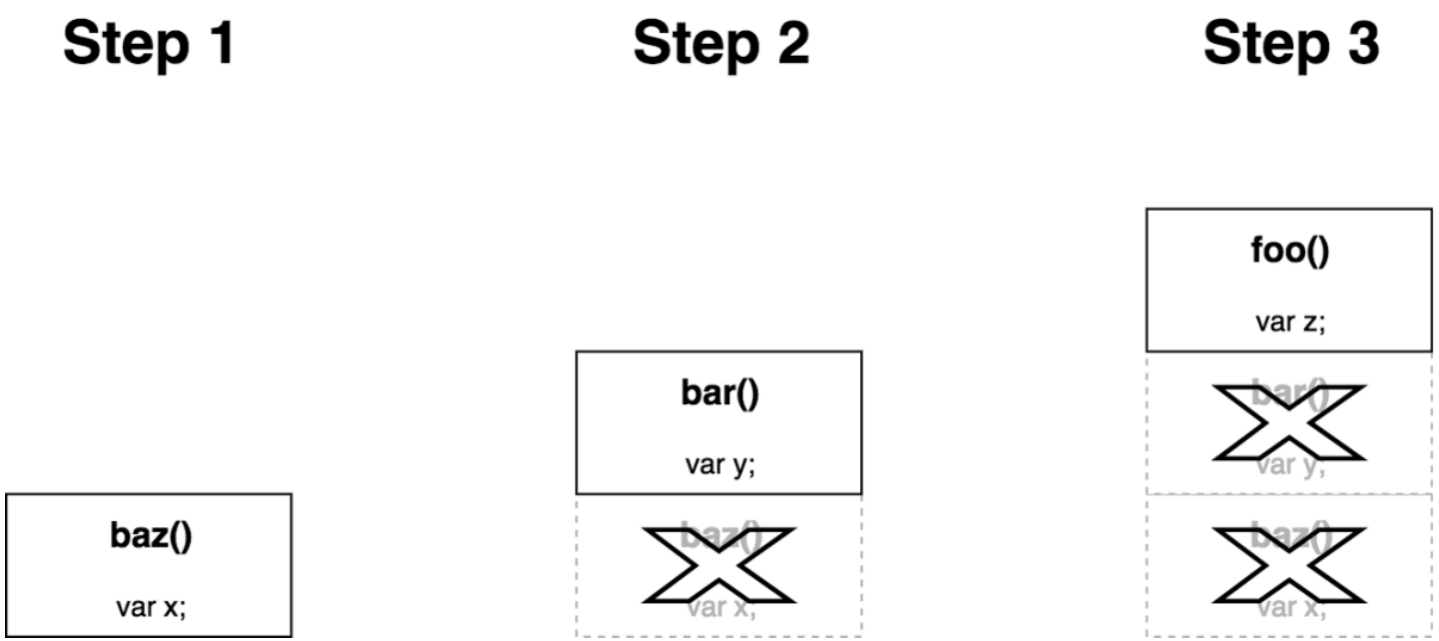
Rekurzija i stek: primer parnih i neparnih

```
function isOdd(v) {
   if (v === 0) return false;
      return isEven(Math.abs(v) - 1 );
}

function isEven(v) {
   if (v === 0) return true;
      return isOdd(Math.abs(v) - 1 );
}
isOdd(33); // => true
isOdd(33333); // RangeError: Maximum call
              // stack size exceeded
```

Repni pozivi: osnovna ideja

- Jedna od fundamentalnih tehnika za ublažavanje problema rekurzije je ***repni poziv***.
- Ideja je sledeća:
  - ako se poziv funkcije `bar()` iz funkcije `baz()` desi na samom kraju izvršenja funkcije `baz()` - što se naziva ***repnim pozivom*** - nema dalje potrebe za aktivacionim zapisom funkcije `baz()`.
  - To znači da se memorija može vratiti, ili, još bolje, jednostavno iskorititi za upravljanje izvršenjem funkcije `bar()`.

Repni poziv u slici



Repni pozivi i rekurzija

- Repni pozivi nisu direktno povezani sa rekurzijom samom za sebe; ova ideja važi za bilo koji funkcijski poziv.
  - Kako je malo je verovatno da će manuelni aktivacioni zapisi ići dublje od 10 nivoa u većini slučajeva, šanse da će oni uticati na ponašanje memorije u programu su male.
- Međutim, repni pozivi “zasijaju punim sjajem” baš u slučaju rekurzije, jer to znači da rekurzivni stek može da raste "doveka", i da jedini problem performanse bude računanje, a ne memorijsko ograničenje.
  - Repna rekurzija može da se izvršava sa složenošću koja odgovara O(1) fiksiranom korišćenju memorije.

Repni pozivi i optimizacija performanse

- Tehnike repnih poziva se često nazivaju i *optimizacije repnih poziva* (engl. *Tail Call Optimizations*, TCO).
- Međutim, tu je važno razlikovati pristup *sposobnosti detektovanja repnog poziva za izvršavanje u fiksnom memorijskom prostoru* od *tehnika koje optimizuju primenu ovoga pristupa*.
- Naime, repni pozivi nisu tehnički optimizacija performanse, kako misle mnogi ljudi, jer se izvršavaju sporije od normalnih poziva.
- Zbog toga, njihovu primenu valja optimizovati, a TCO se baš bavi optimizacijom samih repnih poziva sa ciljem postizanja efikasnijeg izvršavanja koda.

JS podesni repni pozivi

- JavaScript nikada nije zahtevala (niti zabranjivala) repne pozive sve do specifikacije ES6.
- Specifikacija ES6 obavezuje na prepoznavanje repnih poziva specifičnog oblika nazvanih ***podesni repni pozivi*** (***Proper Tail Calls***, PTC) i na garanciju da će taj kod u PTC formi da se izvršava bez neograničenog rasta memorijskog steka.
- Praktično govoreći, to znači da ne bi trebalo da dođe do greške RangeError ako se poštuju zahtevi PTC-a.

JS podesni repni pozivi: zahtevi

- Prvo, PTC u JavaScript-u **zahteva striktni režim**.
- Drugo, poziv funkcije je poslednja stvar koja se izvršava u okružujućoj funkciji i ona mora biti **eksplicitno vraćena** (putem naredbe `return`).
  - Dakle, *prikladan* repni poziv mora da bude poslednja naredba u funkciji i da izgleda ovako:

```
return foo( .. );
```

Nije PTC: Primeri

```
foo();
   return;

// ili

var x = foo( .. );
   return x;

// ili

return 1 + foo( .. );
```

Jeste PTC: Primer

```
return x ? foo( .. ) : bar( .. );
```

Preuređivanje rekurzije

- Za korišćenje rekurzije na problemima koji mogu da dovedu do prekoračenja steka JavaScript endžina potrebno je preurediti rekurzivne pozive tako da se iskoristi PTC ili da se izbegnu ugnježdeni pozivi.
- Postoji više strategija refaktorizacije koje mogu da pomognu, ali i sve one predstavljaju kompromis čega treba biti svestan.
- Mi ćemo prikazati tri tehnike: **zamenu steka**, **prosleđivanje kontinuacije**, i **trambuline**.

Preuređivanje rekurzije: izmena steka

- Ideja:
  - Preurediti korišćenje rekurzije tako da aktivacioni zapis ne mora da se čuva;
  - Izraziti rekurziju u PCT obliku, i
  - Iskoristi optimizovano rukovanje repnim pozivima koje radi JavaScript endžin.
- Primer za demonstraciju (nije PCT jer se, nakon završetka rekurzivnog poziva `sum(...nums)`, varijabla `num1` dodaje na to:

```
function sum(num1,...nums) {
    if (nums.length == 0) return num1;
        return num1 + sum(...nums);
}
```

Izmena steka: pravljenje PTC funkcije

```
function sum(result,num1,...nums) {
// ..
}
```

- Sada treba izvršiti prethodno računanje zbira varijabli `result` i `num1`, i proslediti ga rekurzivnom pozivu:

```
"use strict";
function sum(result,num1,...nums) {
   result = result + num1;
   if (nums.length == 0) return result;
      return sum(result, ...nums );
}
```

Izmena steka: pravljenje PTC funkcije nedostatak

- Sada je `sum()` u PTC formi.
- Ali je nedostatak što je **izmenjena signatura funkcije** na **način koji njeno korišćenje čini neuobičajenim**.
- Pri pozivu se mora, u suštini, proslediti 0 kao prvi argument ispred brojeva koji će se sabirati:

```
sum( /*initialResult=*/0, 3, 1, 17, 94, 8 );
// 123
```

Zamena steka: nova interfejsna funkcija

```
"use strict";
function sumRec(result,num1,...nums) {
    result = result + num1;
   if (nums.length == 0) return result;
      return sumRec(result, ...nums);
}

function sum(...nums) {
   return sumRec( /*initialResult=*/0, ...nums );
}

sum( 3, 1, 17, 94, 8 );    // 123
```

Zamena steka: "sakrivanje" rekurzivne funkcije kao unutrašnje funkcije

```
"use strict";
function sum(...nums) {
   return sumRec( /*initialResult=*/0, ...nums
  );

      function sumRec(result,num1,...nums) {
         result = result + num1;
         if (nums.length == 0) return result;
            return sumRec( result, ...nums );
    }
}

sum( 3, 1, 17, 94, 8 );  // 123
```

Zamena steka: "sakrivanje" rekurzivne funkcije , ali bolje

```
"use strict";
// IIFE funkcijski izraz
var sum = (function(){

   return function sum(...nums) {
      return sumRec( /*initialResult=*/0, ...nums );
   }

   function sumRec(result,num1,...nums) {
      result = result + num1;
      if (nums.length==0) return result;
         return sumRec(result, ...nums);
   }

})();

sum(3, 1, 17, 94, 8);                    // 123
```

Zamena steka: još bolje, ali samo za neke situacije

```
"use strict";

function sum(result,num1,...nums) {
    result = result + num1;
    if (nums.lengt == 0) return result;
        return sum( result, ...nums );
}

sum( /*initialResult=*/0, 3, 1, 17, 94, 8 );
  // 123
```

Zamena steka: još bolje, ali samo za neke situacije

- Samo se preimenuju parametri (`result` u `num1`, `num1` u `num2`):

```
"use strict";

function sum(num1,num2,...nums) {
    num1 = num1 + num2;
    if (nums.length == 0) return num1;
        return sum(num1, ...nums);
}

sum( 3, 1, 17, 94, 8 ); // 123
```

Preuređivanje rekurzije: prosleđivanje kontinuacije

- ***Kontinuacija*** (engl. ***continuation***) označava povratni poziv funkcije koji specificira sledeći(e) korak(e) koji će se izvršiti nakon što određena funkcija završi svoj posao.
- Organizovanje koda na način da svaka funkcija prima drugu funkciju koja će da se izvrši na njenom kraju zove se ***stil prosleđivanja kontinuacije*** (engl. ***Continuation Passing Style***, CPS).

Preuređivanje rekurzije: prosleđivanje kontinuacije

- Neki oblici rekurzije ne mogu se praktično refaktorizovati na čist PTC, posebno višestruka rekurzija.
- Međutim, moguće je izvršiti prvi rekurzivni poziv i obmotati rekurzivne pozive koji slede u funkciju kontinuacije koja će da se prosledi tom prvom pozivu.
  - To neminovno znači mnogo više funkcija koje će se izvršavati na steku. Međutim, memorijski stek neće neograničeno rasti sve dok su sve funkcije, uključujući i kontinuacije, u PTC obliku.

Prosleđivanje kontinuacije: Fibonačijev niz

```
"use strict";

function fib(n, cont = identity) {
    if (n <=1) return cont(n);
    return fib(
    n - 2,
    n2 => fib(
        n - 1,
        n1 => cont(n2 + n1)
          )
    );
}
```

Preuređivanje rekurzije: trambuline

- Još jedna tehnika koja koristi CPS za reorganizaciju rekurzije radi ublažavanja memorijskih zahteva se zove ***trambuline*** (engl. ***Trampolines***).
- Tehnika trambuline takođe kreira CPS-olike kontinuacije, Dok CPS kreira kontinuacije i prosleđuje ih dalje, ova druga tehnika za ublažavanje memorijskog pritiska takođe kreira CPS-olike kontinuacije, ali se one, umesto prosleđivanja, plitko vraćaju.

Preuređivanje rekurzije: helper **trampoline()**

```
function trampoline(fn) {
    return function trampolined(...args) {
        var result = fn( ...args );

        while (typeof result =="function") {
                result = result();
        }

      return result;
    };
}
```

Helper **trampoline()**: korišćenje

```
var sum = trampoline(
   function sum(num1,num2,...nums) {
      num1 = num1 + num2;
      if (nums.length == 0) return num1;
         return () => sum( num1, ...nums );
   }
);

var xs = [];
for (let i = 0; i < 20000; i++) {
    xs.push( i );
}

sum( ...xs );      // 199990000
```

Trambuline naspram CPS

- Bolje performanse u izvršavanju i memoriji nego CPS,
- Manja intruzivnost od CPS u odnosu na deklarativnu rekurzivnu formu - ne zahtevaju izmenu signature funkcije za primanje argumenta kontinuacione funkcije.

Rekurzija i imutabilnost

- Još jedno pitanje zbog koga je važna rekurzija i rekurzivne tehnike odnosi se na njenu vezu sa čistotom i imutabilnošću.
- U tradicionalnim funkcionalnim jezicima, lokalne varijable u stvari i nisu varijable, već su imutabilne vrednosti i ne mogu da se menjaju. OVO NE VAŽI ZA JAVASCRIPT!
- To dovodi do situacije da se lokalna mutacija ne može primeniti, već je potreban poseban mehanizam za te namene.
- Taj mehanizam je modifikovanje vrednosti lokala izmenom steka poziva a to je baš ono što radi rekurzija.

Rekurzija i imutabilnost: sabiranje sa mutacijom lokalnih varijabli

```
/* funkcija summ mutira dve lokalne varijable: i i result. */
function summ(array) {
    var result = 0;
    var sz = array.length;
    for (var i = 0; i < sz; i++)
        result += array[i];
        return result;
}
summ(_.range(1,11));// 55
```

- Ovo radi u JS-u, ali u čistim jezicima ne bi radilo jer u čistim funkcionalnim jezicima lokalne varijable u stvari i nisu varijable, već su imutabilne vrednosti.

Rekurzija i imutabilnost: rekurzivno sabiranje (radi u JS-u, ali i u čistim jezicima)

```
function summRec(array, seed) {
   if (_.isEmpty(array))
      return seed;
   else
      return summRec(_.rest(array),
                     _.first(array) + seed);
}
summRec([], 0); // 0
summRec(_.range(1,11), 0); // 55
```

Rekurzivne strukture podataka

- U jezicima računarskog programiranja, rekurzivni tip podataka (rekurzivno definisan, induktivno definisan ili induktivni tip podataka) je tip podataka za vrednosti koje mogu da sadrže druge vrednosti istog tipa.
- Podaci rekurzivnih tipova se obično posmatraju kao usmereni grafovi.
- Važna primena rekurzije u računarstvu je u definisanju dinamičkih struktura podataka kao što su liste i stabla.
- Rekurzivne strukture podataka mogu dinamički da rastu do proizvoljne veličine kao odgovor na zahteve u toku izvršavanja, dok zahtevi za veličinu statičkog niza moraju biti postavljeni u vreme kompajliranja.

Rekurzivni tipovi: direktna rekurzija

- Tip `List` u jeziku Haskell koji kaže: lista **a** je ili prazna lista ili cons ćelija koja sadrži **a** („glavu“ liste) i drugu listu („rep“).

```
data List a = Nil | Cons a (List a)
```

Drugi primer je sličan jednostruko povezan tip u Java-i koji kaže: neprazna lista tipa E sadrži podatak tipa E i referencu na drugi objekat `List` za ostatak liste (ili null referencu koja označava da je ovo kraj liste).

```
class List<E> {
    E value;
    List<E> next;
}
```

Rekurzivni tipovi: uzajamna rekurzija

- Tipovi podataka se mogu definisati i uzajamnom rekurzijom.
- Najvažniji osnovni primer je stablo, koje se može definisati uzajamno rekurzivno u terminima šume (`f`) (lista stabala (`t`)). Simbolički to izgleda ovako:

```
f: [t[1], ..., t[k]]
t: v f
```

- Šuma `f` se sastoji od liste stabala, dok se stablo `t` sastoji od para koji čine vrednost `v` i šuma `f` (njena deca). Dakle, stablo je izraženo jednostavnim terminima: lista jednog tipa i par dva tipa.
- U jeziku Haskell to izgleda ovako:

```
data Tree a      = Empty | Node (a, Forest a)
data Forest a    = Nil | Cons (Tree a) (Forest a)
```

Rekurzija: sažetak

- Direktna rekurzija je funkcija koja sebe poziva najmanje jednom i dispečira pozive na sebe sve dok se ne zadovolji bazni uslov. Višestruka rekurzija (kao što je, na primer, binarna rekurzija) je kada funkcija samu sebe poziva više puta. Uzajamna rekurzija je kada dve ili više funkcija uzajamno pozivaju jedna drugu rekurzivno u petlji.
- Prednost rekurzije je što je ona deklarativnija i zbog toga čitljivija. Nedostatak je performansa (brzina izvršavanja), a još više memorijska ograničenja.
- Terminalni pozivi ublažavaju memorijski pritisak ponovnim korišćenjem/poništavanjem stek frejmova. JavaScript zahteva striktni režim i prikladne terminalne pozive (PTC) da bi se iskoristila prednost ove "optimizacije". Ima više tehnika koje se mogu koristiti pojedinačno ili kombinovano za refaktorisanje ne-PTC rekurzivne funkcije u PTC formu ili, u najmanju ruku, za izbegavanje memorijskih ograničenja "tanjenjem" steka.
- **Zapamtite:**
- REKURZIJU TREBA KORISTITI DA BI SE **NAPRAVIO ČITLJIVIJI KOD**. AKO SE **KORISTI NA POGREŠAN NAČIN** ILI SE ZLOUPOTREBLJAVA**, ČITLJIVOST ĆE BITI LOŠIJA NEGO KOD IMPERATIVNE FORME**. **NEMOJTE TO DA RADITE!**

Memoizacija

- Šta je memoizacija
- Zašto memoizacija
- Kako memoizacija

Šta je memoizacija

- Memoizacija je tehnika za optimizaciju računanja time što će eliminisati višestruka računanja.
- Ideja je da se “skupa” računanja (ona koja dugo traju i/ili zahtevaju puno računarskih resursa) rade samo jednom i negde zapamte tako da, kad god programu zatreba rezultat tih računanja, on neće morati da ih ponovo izvršava, već će da koristi prethodno zapamćene rezultate.
- Dakle, memoizacija je tehnika za keširanje rezultata sa ciljem ubrzavanja računarskih programa.

Zašto memoizacija

- Funkcije su vitalni gradivni blokovi svakog programa.
- Funkcije se u programu najčešće pozivaju više puta za zadate vrednosti argumenata.
- Često, dešava se da se zahteva pozivanje funkcije više puta za iste vrednosti argumenata
- U takvim slučajevima (ako se ništa ne preduzme), troši se vreme i procesni resursi za sračunavanja koja su već ranije bila urađena
- Blisko je zdravom razumu da se to izbegne, naročito kada su sračunavanja dugotrajna i troše puno procesnih resursa.

Kako radi memoizacija

- Kada se pozove funkcija, memoizacija negde uskladišti rezultat poziva pre no što kontrolu programa vrati kodu koji je funkciju pozvao.
- To omogućuje da se ta uskladištena vrednost pri sledećem pozivanju sa istim argumentima direktno preuzme iz skladišta, umesto da se ponovo izvršava sama funkcija.
- Na taj način memoizacijom se izbegavaju nepotrebna izvršavanja funkcije i time se može značajno uštedeti na vremenu (brže se dobavlja vrednost iz skladišta nego što je izvršavanje funkcije) i angažovanju procesora (nema računanja).

Memoizacija i rekurzija

- Memoizacija je pogodna za optimizaciju rekurzivnih računanja jer se u značajnom broju situacija javljaju višestruka pozivanja funkcije sa istim argumentom/argumentima.
- Primer koji se najčešće koristi je Fobonacci-jev niz gde je rekurzivna formula sledećeg oblika

  Za svako celo $n > 1$,

  $Fib(n) = \boldsymbol{Fib(n-1)} + \boldsymbol{Fib(n-2)}$

Opet Fibonacci

- Bez memoizacije:

```
function fibonacci(n) {
  if (n === 0 || n === 1)
    return n;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Opet Fibonacci

- Sa memoizacijom:

```
var fibonaccimemo = (function() {
  var memo = {};

  function f(n) {
    var value;

    if (n in memo) {
      value = memo[n];

    } else {
      if (n === 0 || n === 1)
        value = n;

      else
        value = f(n - 1) + f(n - 2);
      memo[n] = value;
    }

    return value;
  }

  return f;
})();
```

Probajte, ali ne za n >45

```
n = prompt(' Unesite n: ', 35);

// sa memoizacijom
var startTimememo = Date.now();
console.log ( 'Startno vreme ',startTimememo );
var fibResmemo = fibonaccimemo(n);
var endTimememo = Date.now();
console.log ( 'Vreme završetka ',endTimememo );

var durationmemo = endTimememo - startTimememo;
console.log ( 'Za n = ', n, 'Fibonacci sa memoizacijom = ', fibResmemo);
console.log ( 'Trajanje (milisekundi) = ', durationmemo);

// Bez memoizacije
var startTimenomemo = Date.now();
console.log ( 'Startno vreme ',startTimenomemo );
var fibResnomemo = fibonacci(n);
var endTimenomemo = Date.now();
console.log ( 'Vreme završetka ',endTimenomemo );
var durationnomemo = endTimenomemo - startTimenomemo;
console.log ( 'Za n = ', n, 'Fibonacci bez memoizacije = ',
fibResnomemo);
console.log ( 'Trajanje (milisekundi) = ', durationnomemo);
```

Rezultat za n=35

Startno vreme 1699961860112

Vreme završetka 1699961860112

Za n = 35 Fibonacci sa memoizacijom = 9227465

Trajanje (milisekundi) = **0**

Startno vreme 1699961860112

Vreme završetka 1699961860230

Za n = 35 Fibonacci bez memoizacije = 9227465

Trajanje (milisekundi) = **118**

Više argumenata

- Primer koji smo videli ima samo jedan argument (n)
  - Rukovanje kešom je jednostavno – keš je jedan objekat a vrednost argumenta je svojstvo tog objekta
- Realan život je višedimenzionalan: Rukovanje kešom se komplikuje.
- Da bi se memoizovala funkcija sa više argumenata, keš mora da bude višedimenzionalan, ili se svi argumenti moraju kombinovati u jedan indeks.

Višedimenzionalni keš

- Keš više nije jedan objekat, već postaje hijerarhija objekata
- Svaka dimenzija se indeksira po jednom parametru/argumentu.

Višedimenzionalni keš: Fibonačijeva funkcija

```
var fibonacci = (function() {
  var memo = {};

  function f(x, n) {
    var value;

    memo[x] = memo[x] || {};

    if (x in memo && n in memo[x]) {
      value = memo[x][n];
    } else {
      if (n === 0 || n === 1)
        value = n;
      else
        value = f(x, n - 1) + f(x, n - 2);

      memo[x][n] = value;
    }

    return value;
  }

  return f;
})();
```

Visedimenzionalni keš: primer poziva

```
// Primer poziva
let x = 12
let y = 13
let z = 14
console.log ('x=', x, fibonacci('x',x))
console.log ('y=', y, fibonacci('y',y))
console.log ('z=', z, fibonacci('z',z))
```

Jednodimenzionalni keš za više argumenata

- Ovde se argumenti transformišu u niz (tip `Array`) i zatim se koriste za indeksiranje keša.
- Za to se može koristiti objekat `arguments` koji se mora transformisati u tip `Array` .
- Nakon transformacije, keširanje se radi na prethodno opisani način.
- Transformacija se može uraditi korišćenjem metode `slice()`.

Primer: Transformacija `arguments` u niz

```
var fibonacci = (function() {
  var memo = {};
  var slice = Array.prototype.slice;

  function f(x, n) {
    var args = slice.call(arguments);
    var value;

    if (args in memo) {
      value = memo[args];
    } else {
      if (n === 0 || n === 1)
        value = n;
      else
        value = f(x, n - 1) + f(x, n - 2);

      memo[arguments] = value;
    }

    return value;
  }

  return f;
})();
// Primer poziva
let argumenti = ['x', 'y', 'z']
console.log (fibonacci (argumenti, 12))
```

Šta kada su argumenti objekti

- Prikazana šema ne radi dobro sa objektima
- Razlog je što se objekti konvertuju u string “[object Object]” pa se više objekata (a oni su ovde indeksi) preslikava na istu lokaciju keša.
- To se može popraviti stringifikacijom objektnih argumenata pre indeksiranja.
- Naravno, to će da uspori memoizaciju.

Primer: memoizacija sa objektnim argumentom

```
var foo = (function() {
  var memo = {};

  function f(obj) {
    var index = JSON.stringify(obj);

    if (index in memo) {
      return memo[index];
    } else {
      // memoizacija funkcije
      return (memo[index] = function_value);
    }

  }

  return f;
})();
```

Automatska memoizacija

```
function memoize(func) {
  var memo = {};
  var slice = Array.prototype.slice;

  return function() { // nova funkcija koja se vraća
    var args = slice.call(arguments);

    if (args in memo)
      return memo[args];
    else
      return (memo[args] = func.apply(this, args));

  }
}
```

Ograničenja pri memoizaciji

- **REFERENCIJALNA TRANSPARENTNOST je obavezan uslov za (automatsku) memoizaciju!!!**

```
var bar = 1;

function foo(baz) {
  return baz + bar;
}

console.log (' Poziv foo(1) pre poziva bar++: ',foo(1));// 2
bar++;
console.log (' Poziv foo(1) nakon poziva bar++: ',foo(1));// 3
```

- Konzumiranje memorije

Memoizacija: Sažetak

- Može da poveća performanse.
- Skladišti izračunate vrednosti funkcije u keš koji se indeksira argumentima
  - Ako argument postoji u kešu, uzima se uskladištena vrednost.
  - Ako ne postoji, računa se vrednost i skladišti se u keš.
- Objektne argumente treba prethodno stringifikovati.
- Može se automatski primeniti samo na referencijalno transparentne funkcije.
- Nije potrebna za funkcije koje se retko pozivaju, niti za funkcije koje se brzo izvršavaju.

Literatura za predavanje

1. Z. Konjović, Funkcionalno programiranje, **Tema 05 – Rekurzija**, slajdovi sa predavanja, dostupni na folderu **FP kurs 2023-24→Files →Slajdovi sa predavanja**
2. M. Fogus, Functional JavaScript, O'Reilly Media, Inc., 2013, Poglavlje 6 "**Recursion**"