Funkcionalno programiranje

Elementi jezika Scala

Računanje vrednosti izraza

- Složen izraz računa se repetitivno na sledeći način:
  - polazi se od najlevlje operacije
  - računaju se vrednosti operanada
  - primenjuje se operator na vrednosti operanada

- **Definisano ime** se računa zamenom imena desnom stranom definicije

- Postupak računanja vrednosti se zaustavlja kada se dostigne **vrednost** (pr: numerička vrednost)

Računanje vrednosti izraza -primer-

REDUKCIJA: Postupak simplifikacije izraza u vrednosti (korak po korak)

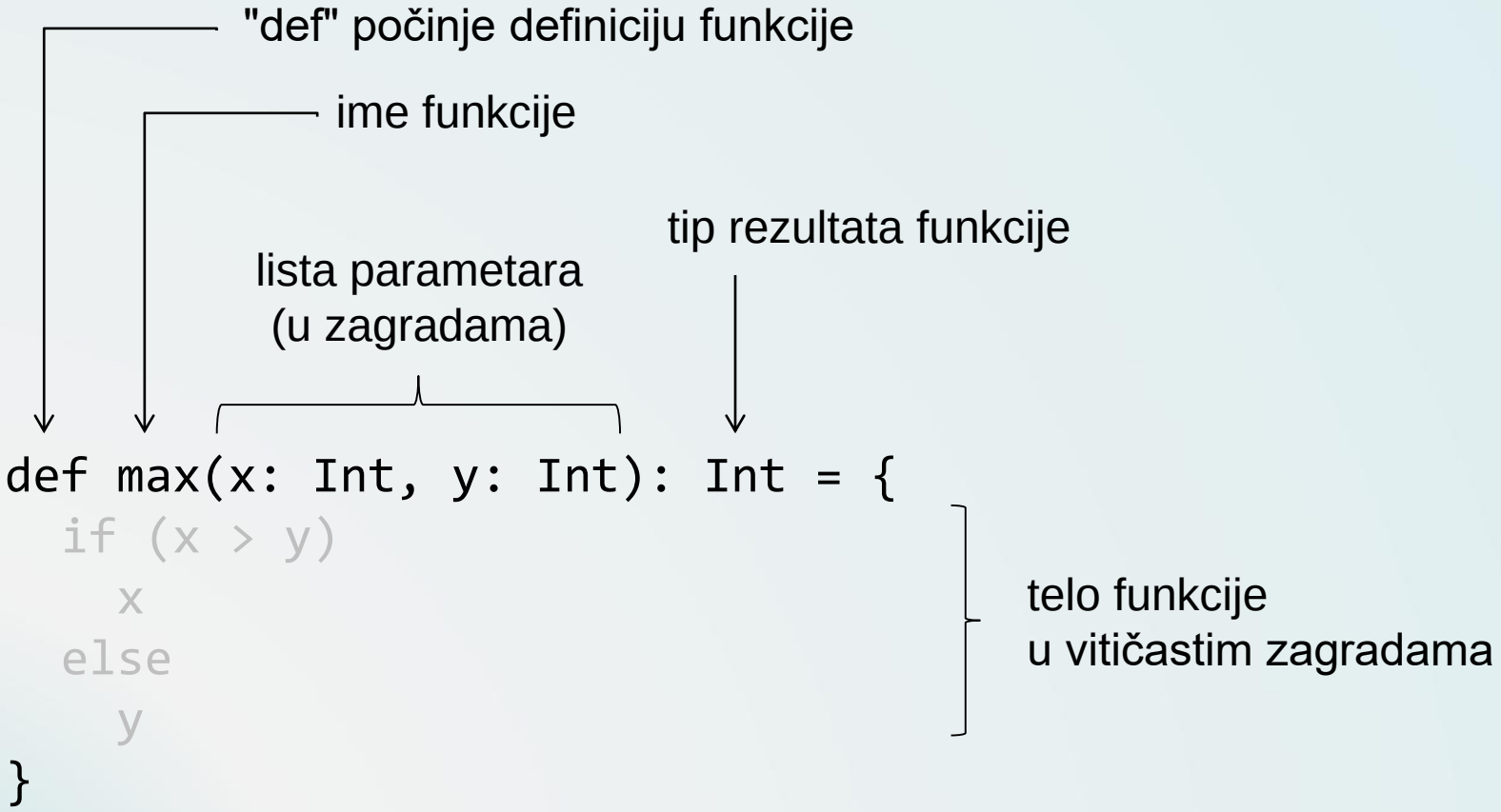
```
def pi = 3.14159
def radius = 10

      (2 * pi) * radius
→     (2 * 3.14159) * radius
→     6.28318 * radius
→     6.28318 * 10
→     62.8318
```

Definisanje funkcija

```
scala> def square(x : Double) = x * x
square : (Double)Double
scala> square(2)
4.0
scala> square(5 + 4)
81.0
scala> square(square(4))
256.0
scala> def sumOfSquares(x:Double,y:Double)=square(x)+square(y)
sumOfSquares : (Double,Double)Double
```

Definisanje funkcija



Saglasno tome: `def pi = 3.14159`

je definicija funkcije bez parametara konstantne vrednosti

Definisanje funkcija

- U svakoj funkciji mogu se definisati druge funkcije kao bilo koja druga vrsta podataka
- Ugneždene funkcije su lokalne za okružujuću
- Formalni argumenti (parametri) okružujuće funkcije **vidljivi su** u ugneždenim
  - korisno kada se parametar okružujuće funkcije koristi u ugneždenim, jer ga zapravo nije potrebno prenositi

- Sintaksni detalj: upotreba separatora iskaza ;
  - kompajler tumači kraj linije kao **;** osim ako:
    - linija se završava nekom reči koja nije validan kraj iskaza
    - naredna linija počinje rečju kojom ne može da počne iskaz
    - linija se završava "unutar" zagrada ( ) ili [ ]

Računanje vrednosti funkcija

- Vrednost funkcije (sa parametrima) određuje se na isti način kao i kod izraza:
  - računa se vrednost svih argumenata funkcije (s leva udesno)
  - poziv funkcije se zamenjuje desnim delom definicije funkcije
  - formalni argumenti (parametri) se zamenjuju stvarnim argumentima
- Model računanja zamenom (supstitucijom)

```
    sumOfSquares(3, 2+2)
→   sumOfSquares(3, 4)
→   square(3) + square(4)
→   3 * 3 + square(4)
→   9 + square(4)
→   9 + 4 * 4
→   9 + 16
→   25
```

Poziv po vrednosti i po imenu

- U prethodnom primeru:
  - argumenti funkcija se redukuju u vrednosti pre supstitucije
  - takav način računanja vrednosti zove se **poziv po vrednosti**

- Moglo bi obrnuto: izvršiti supstituciju pre redukcije
  - takav način računanja vrednosti zove se **poziv po imenu**

```
    sumOfSquares(3, 2+2)
→   square(3) + square(2+2)
→   3 * 3 + square(2+2)
→   9 + square(2+2)
→   9 + (2+2) * (2+2)
→   9 + 4 * (2+2)
→   9 + 4 * 4
→   9 + 16
→   25
```

Poziv po vrednosti i po imenu

- Poziv po vrednosti
  - izbegava ponovljena računanja vrednosti argumenata
- Poziv po imenu
  - izbegava računanje vrednosti argumenata kada se oni ne koriste u funkciji
- Računanje pozivom po imenu uvek rezultira vrednošću ako poziv po vrednosti rezultira vrednošću
- Obrnuto nije uvek tačno!

Poziv po vrednosti i po imenu

- Primer

```
scala> def loop: Int = loop
loop: => Int

scala> def first(x: Int, y: Int) = x
first: (Int,Int)Int
```

Poziv po imenu

first(1, loop)

→ 1

Poziv po vrednosti

first(1, loop)

→ *first(1, loop)*

→ *first(1, loop)*

→ *first(1, loop)*

...

U jeziku Scala, podrazumeva se **poziv po vrednosti.**

Poziv po vrednosti i po imenu

- Da bi se proizveo poziv po imenu, tipu parametra treba da prethodi **=>**

```
scala> def constOne(x : Int, y : => Int) = 1
constOne: (Int,=>Int)Int

scala> constOne(1, loop)
1

scala> constOne(loop, 2)
^C
```

beskonačan ciklus

Uslovni izrazi

- Sintaksa ista kao u jeziku Java: *if-else*
- Tumačenje je kao za operator u jeziku Java: **... ? ... : ...**

```
scala> def abs(x: Double) = if(x>=0) x else -x
abs: (Double)Double

scala> def max(x : Int, y : Int) = if(x>y) x else y
max: (x: Int, y: Int)Int
```

- **if** koristi predikat logičkog (Bulovog) tipa
- operatori
  - &&
  - ||
  - !

Uslovni izrazi - primer

- Definisati operatore && i || pomoću **if**

```
def and(x: Boolean, y: Boolean) =
   if(x) y else False

def or(x: Boolean, y: Boolean) =
   if(x) x else y
```

Blokovi, iskazi i leksički doseg

- **Definicija** i **izraz** (expression)
  su dve vrste **iskaza** (statement)
- Blok se sastoji od definicija i izraza
- Poslednji izraz u bloku predstavlja **rezultat bloka**
- Blok je izraz – dakle može da se projavi
  gde god se očekuje izraz
- Iza svakog iskaza u bloku obavezno dolazi znak **;** izuzev
  - ako sledi novi red
  - ako je iskaz poslednji, neposredno pre pre znaka }
- Imena definisana u okružujućem bloku su vidljiva
  i u unutrašnjim blokovima (izuzev ako su sakrivena)

Terminalna rekurzija (tail recursion)

- Posmatra se definicija funkcije za određivanje najvećeg zajedničkog delioca

```
def gcd(a: Int, b: Int): Int = if(b==0) a else gcd(b, a%b)
```

- Evaluacija gcd(14, 21) bi izgledala ovako:

```
gcd(14, 21)
→       if (21 == 0) 14 else gcd(21, 14 % 21)
→       if (false) 14 else gcd(21, 14 % 21)
→       gcd(21, 14 % 21)
→       gcd(21, 14)
→       if (14 == 0) 21 else gcd(14, 21 % 14)
→ →     gcd(14, 21 % 14)
→       gcd(14, 7)
→       if (7 == 0) 14 else gcd(7, 14 % 7)
→ →     gcd(7, 14 % 7)
→       gcd(7, 0)
→       if (0 == 0) 7 else gcd(0, 7%0)
→ →     7
```

Terminalna rekurzija (tail recursion)

- Posmatra se definicija funkcije za računanje faktorijela

```
def fact(n: Int): Int = if(n==0) 1 else n*fact(n-1)
```

- Evaluacija fact(5) bi izgledala ovako:

```
fact(5)
→       if (5 == 0) 1 else 5*fact(5-1)
→...    5*fact(5-1)
→...    5*fact(4)
→...    5*(4*fact(3))
→...    5*(4*(3*fact(2)))
→...    5*(4*(3*(2*fact(1))))
→...    5*(4*(3*(2*(1*fact(0)))))
→       5*(4*(3*(2*(1*1))))
→...    120
```

Postoji li razlika u načinu računanja vrednosti gdc(14, 21) i fact(5)?

Terminalna rekurzija (tail recursion)

- Ako funkcija poziva samu sebe u svom poslednjem izrazu, aktivacioni zapis (stack frame) ove funkcije može biti iskorišćen u rekurzivnom pozivu
- To se zove **terminalna rekurzija**
- Funkcija `gcd` se suštinski nikad ne vraća iz rekurzije
- Funkcija `fact` mora da izvrši povratak, da bi se obavilo množenje sa n

Terminalna rekurzija (tail recursion)

- Funkcije sa terminalnom rekurzijom svode se na **iterativni** postupak
  - može biti približno podjednako efikasna kao i iteracija (ciklus)
- Generalno: ako se poslednja aktivnost funkcije sastoji od poziva neke funkcije (može biti isti entitet u pitanju), jedan aktivacioni zapis na steku je dovoljan za obe

xkcd.com

Terminalna rekurzija (tail recursion)

- Anotacija `@tailrec` (`scala.annotation`)

```
scala> import scala.annotation.tailrec
import scala.annotation.tailrec

scala> @tailrec
     | def gcd(a: Int, b: Int): Int = if(b==0) a else gcd(b, a%b)
gcd: (a: Int, b: Int)Int

scala> @tailrec
     | def fact(n: Int): Int = if(n==0) 1 else n*fact(n-1)
<console>:13: error: could not optimize @tailrec annotated method
fact: it contains a recursive call not in tail position
```

Definicija vrednosti

- (podsećanje) Definicija funkcije bez parametara glasi:
  def *ime* = *izraz*
  - *izraz* se izračunava svakog puta kada se koristi *ime*
- Definicija vrednosti glasi:
  val *ime* = *izraz*
  - time se uvodi ime za **vrednost** izraza *izraz*
  - *izraz* će biti izračunat samo jednom, u trenutku definicije vrednosti
  - kao i kod funkcije: tip vrednosti je neophodan
    samo ako se ne može zaključiti iz izraza
- Primer
  - `val x = 2` `// x: Int = 2`
  - `val y = square(x)` `// y: Int = 4`
  - `def loop: Int = loop` `// loop: Int`
    `val x: Int = loop` `// ... beskonačan ciklus`

Promenljive

- Vrednosti mogu biti definisane na dva načina
  - pomoću **val**: tada se imenu ne može dodeliti nova vrednost, ponaša se kao final podatak u Javi
  - pomoću **var**: tada se imenu može dodeliti nova vrednost

```
scala> val msg: String = "Hello, world!" // ili:   val msg = ...
msg: String = Hello, world!

scala> msg = "Goodbye cruel world!"
<console>:6: error: reassignment to val
msg = "Goodbye cruel world!"

scala> var greeting = "Hello, world!"
greeting: java.lang.String = Hello, world!

scala> greeting = "Leave me alone, world!"
greeting: java.lang.String = Leave me alone, world!
```

Preporuke: val ili var?

- Favorizovati
  - **val**, **nepromenljive** (immutable) objekte
  - funkcije (metode) **bez bočnih efekata**

- Koristiti **var**, promenljive (mutable) objekte i funkcije (metode) sa bočnim efektima kada postoji opravdana potreba za njima

- Jedan pokazatelj da li se programira u imperativnom stilu jeste upotreba **var**. U funkcionalnom stilu **var** nije poželjan
  - ipak, Scala ostavlja mogućnost da programer kombinuje stilove

- Parametri funkcija su **val**

Promenljivi naspram nepromenljivih objekata

- Prednosti nepromenljivih
  - jednostavniji za rezonovanje
    (ne prolaze kroz promenu stanja u toku rada programa)
  - jednostavniji za manipulaciju,
    prenose se bez potrebe za pravljenjem (defanzivnih) kopija
  - jednostavnija upotreba u višenitnom okruženju,
    ne postoji način da više istovremenih pristupa pokvari stanje objekta
  - bezbedno heširanje, ako se ključ određuje na osnovu stanja

- Mane nepromenljivih
  - zahtevaju obimno kopiranje umesto jednostavnog ažuriranja
    - zato biblioteke često nude promenljive alternative

Preporuke u vezi stila

- Imperativni stil

```
def printArgs(args: Array[String]): Unit = {
   var i = 0
   while (i < args.length) {
        println(args(i))
        i += 1
   }
}
```

- Funkcionalni stil (ali nije čist funkc. stil – ima bočni efekat)

```
def printArgs(args: Array[String]): Unit =
{  for (arg <- args)  println(arg)   }
```

- Ili

```
def printArgs(args: Array[String]): Unit =
{  args.foreach(println)     }
```


http://docs.scala-lang.org/style/declarations.html

Funkcije višeg reda

- Funkcijski jezici tretiraju funkcije kao "građane prvog reda"
  - funkcija može biti prosleđena kao parametar drugoj funkciji
  - funkcija može biti rezultat druge funkcije
- Takav pristup omogućava
  veoma fleksibilan način sastavljanja programa
- Funkcije koje
  - primaju druge funkcije kao parametre ili
  - koje vraćaju druge funkcije kao rezultat

  zovu se **funkcije višeg reda**

Funkcije višeg reda

- Primer: sabrati sve cele brojeve između *a* i *b* ($a \leq b$)

```
def sumInts(a: Int, b: Int): Int =
  if( a > b ) 0 else a + sumInts(a+1, b)
```

- Primer: sabrati kubove svih celih brojeva između a i b

```
def cube(x: Int) = x * x * x
def sumCubes(a: Int, b: Int): Int =
  if( a > b ) 0 else cube(a) + sumCubes(a+1, b)
```

- Može se primetiti da su ova dva primera spec. slučajevi

$$\sum_{n=a}^{b} f(n)$$

za različite vrednosti `f(n)`.

Cilj je formulisati zajedničko ponašanje

Funkcije višeg reda

- Definisati funkciju višeg reda, koja vrši sumiranje

```
def sum(f: Int => Double, a: Int, b: Int) : Double = {
  if (a > b )     0
  else f(a) + sum(f, a+1, b)
}
```

- Definisati funkcije koje se primenjuju nad pojedinačnim članovima sumacionog opsega

```
def id(x: Int) : Double = x
def cube(x: Int) : Double = x*x*x
```

- Onda je moguće napisati:

```
def sumInts(a: Int, b: Int): Double = sum(id, a, b)
def sumCubes(a: Int, b: Int): Double = sum(cube, a, b)
```

Funkcije višeg reda

- U definiciji

```
def sum(f: Int => Double, a: Int, b: Int)
```

`Int => Double` je tip funkcija:

  - koje primaju parametar tipa `Int`
  - čiji rezultat je tipa `Double`
  - drugim rečima: funkcije koje preslikavaju `Int u Double`

- U slučaju više parametara, potrebno je staviti zagrade
  - primer: `(Int, Int) => Double`
  - primer: `(Int => String, Int) => Double`

Funkcije višeg reda: Anonimne funkcije

- Motivacija
  - funkcije kao parametri zahtevaju pisanje (potencijalno) velikog broja kratkih funkcija
  - ponekad je opterećenje za programera da definiše i imenuje sve te funkcije (koristeći **def**)
- Rešenje – anonimne funkcije (funkcijski literali)

```
(x: Int) => x*x*x
```

- Tip parametra može biti izostavljen ako prevodilac može da ga dedukuje (iz konteksta)
- Funkcijski literal se prevodi u klasu
  - instanciranje ove klase u vreme izvršenja daje *funkcijsku vrednost*
  - funkcijske vrednosti su **objekti** (mogu se referisati) i **funkcije** (mogu se pozivati na način uobičajen za funkcije)

Funkcije višeg reda: Funkcijske vrednosti i zatvaranje

1) `(x: Int) => x+1  // x je vezana promenljiva`
2) `(x: Int) => x+y  // y je slobodna promenljiva`

- Anonimna funkcija (funkcijski literal):
  - sa slobodnim promenljivim zove se otvoren član (*open term*)
  - bez slobodnih promenljivih zove se zatvoren član (*closed term*)
- Zatvaranje (closure) je funkcijska vrednost koja "zahvata" veze ka slobodnim promenljivim
  - zatvaranje zahvata slobodne promenljive, odnosno od otvorenog pravi zatvoren član
- Zatvaranje ne postaje zatvoren član striktno govoreći, jer je već dato u zatvorenoj formi

Funkcije višeg reda: Funkcijske vrednosti i zatvaranje

- Postavlja se pitanje: kom objektu pristupa zatvaranje?
- Zatvaranje pristupa objektu referisanom putem slobodne promenljive u trenutku nastanka zatvaranja
- Primer:

```
def makeIncreaser(more: Int) = (x: Int) => x + more
scala> val inc1 = makeIncreaser(1)   // more = 1
inc1: (Int) => Int = <function1>
scala> val inc9999 = makeIncreaser(9999)    // more = 9999
inc9999: (Int) => Int = <function1>
```

- Iako je `more` lokalna promenljiva funkcije makeIncreaser, njen životni vek nije ograničen tom funkcijom (automatski se stavlja na hip, umesto na stek)

Funkcije višeg reda: Anonimne funkcije

- Samo sintaksna pogodnost (nisu neophodni deo jezika)
  - anonimna funkcija `(x1: T1, ..., xn: Tn) => E` može da se napiše kao:
  - `{ def f(x1: T1, ..., xn: Tn) = E; f _ }`
    `// ime f se ne koristi u delu programa koji prethodi`

Parcijalno primenjena funkcija (biće objašnjeno kasnije)

- Kako su anonimne funkcije realizovane? **Anonimne klase**

```
(x: Int) => x+1
```

```
new Function1[Int, Int] {
   def apply(x: Int): Int = x+1
}
```

```
(x: Int, y: Int) =>
   "(" + x + ", " + y + ")"
```

```
Function2[Int, Int, String]
```

```
() =>
{ System.getProperty("...") }
```

```
Function0[String]
```

Funkcije višeg reda: Anonimne funkcije

- Sumacija primenom anonimnih funkcija
  - ```
    def sumInts(a: Int, b: Int): Double =
        sum(x=>x, a, b)
    ```
  - ```
    def sumCubes(a: Int, b: Int): Double =
        sum(x=>x*x*x, a, b)
    ```
- Da li su a i b neophodni u funkciji sum?
  - a i b su parametri funkcija `sumInts` i `sumCubes` samo da bi se preneli u funkciju `sum`
  - da li se može napisati funkcija `sum` tako da ne prima a i b kao parametre?

Funkcije višeg reda: kerifikacija (curryfication)

- Poreklo naziva: Haskell Brooks Curry (1900-1982)
  - američki matematičar i logičar
  - razvio koncept kombinatorne logike
  - 3 programska jezika: Haskell, Brook, Curry
  - "kerifikacija" (Gottlob Frege, Moses Schönfinkel)

- Ideja: funkcija više parametara može se predstaviti u vidu sekvence (lanca) funkcija jednog parametra

Funkcije višeg reda: kerifikacija

- Nova definicija funkcije `sum`:

```
def sum(f: Int => Double): (Int, Int) => Double = {
  def sumF(a: Int, b: Int): Double =
    if ( a>b )  0
    else        f(a) + sumF(a+1, b)
  sumF
}
```

- Sada je `sum` postala funkcija čiji rezultat je druga funkcija
  - parametar `f` je "konfigurisao" / "specijalizovao" rezultujuću f-ju
  - rezultujuća funkcija prima dva parametra tipa `Int`
- Sada je moguće definisati:
  - `def sumInts = sum(x => x)`
  - `def sumCubes = sum(x => x*x*x)`

Funkcije višeg reda: kerifikacija

```
def sum(f: Int => Double): (Int, Int) => Double = {
  def sumF(a: Int, b: Int): Double =
    if ( a>b )    0
    else          f(a) + sumF(a+1, b)
  sumF
}
def sumInts = sum(x => x)
def sumCubes = sum(x => x*x*x )
```

- Na osnovu toga, moguće je napisati

```
scala> sumCubes(1, 10) + sumInts(10, 20) //res0: Double = 3190.0
scala> sum(x => -x)(2,3)  // res1: Double = -5.0
scala> def half(x: Int): Double = x/2.0
scala> sum(half)(1,10)    // res2: Double = 27.5
```

sum(half)(1,10) → (sum(half))(1, 10)

Funkcije višeg reda: kerifikacija

- Funkcije koje kao rezultat daju druge funkcije se veoma često koriste u funkcionalnom programiranju
  - zato u jeziku Scala postoji posebna sintaksa
  - prethodna definicija sum bi se skraćeno napisala

```
def sum(f : Int => Double)(a : Int, b : Int): Double =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```

- Iz prethodnog primera, moguće je izvesti sledeće pravilo (bez dokaza)
  - kerifikovana definicija:
    $\texttt{def f (args}_1\texttt{) ... (args}_n\texttt{) = E}$
  - ekvivalentna je definiciji:
    $\texttt{def f (args}_1\texttt{) ... (args}_{n-1}\texttt{) = \{ def g (args}_n\texttt{) = E ; g \}}$

Funkcije višeg reda: kerifikacija

- kerifikovana definicija:
  `def f (args`$_1$`) ... (args`$_n$`) = E`
- ekvivalentna je definiciji:
  `def f (args`$_1$`) ... (args`$_{n-1}$`) = { def g (args`$_n$`) = E ; g }`

- to se može skraćeno zapisati:
  `def f (args`$_1$`) ... (args`$_{n-1}$`) = args`$_n$ `=> E`

- kada se postupak primeni potreban broj puta, dobija se
  `def f = (args`$_1$`) => (args`$_2$`) => ... => (args`$_n$`) => E`
  - primetiti: asocijativnost je s desna ulevo
  - `def f4=(x:Int)=>(y:Int)=>x*y // Int=>(Int=>Int)`

- Ovakav način definisanja i pozivanja funkcija zove se **kerifikacija** (curryfication)

Funkcije višeg reda: kerifikacija

```
scala> def f4=(x:Int)=>(y:Int)=>x*y
f4: Int => (Int => Int)
scala> f4(2)(3)
res5: Int = 6
scala> f4
res6: Int => (Int => Int) = $$Lambda$1064/1507222570@5dfc2a4


scala> def f5(x:Int)(y:Int)=x*y
f5: (x: Int)(y: Int)Int
scala> f5(2)(3)
res7: Int = 6
scala> f5
<console>:13: error: missing argument list for method f5
scala> f5 (_)
res8: Int => (Int => Int) = $$Lambda$1072/1159206653@56e5c8fb
```

Rezime

- U funkcionalnom programiranju, funkcije su osnovne apstrakcije
- Apstrakcije mogu biti kombinovane i mogu se stvarati nove apstrakcije zahvaljujući funkcijama višeg reda
- Treba voditi računa da se iskoriste mogućnosti apstrakcije i višestruke upotrebe koda
- Najviši nivo apstrakcije nije uvek najbolji, ali treba poznavati tehnike apstrakcije da bi se primenile kada to ima smisla

Rezime – elementi jezika

- Tipovi (EBNF notacija):

  Type = SimpleType | FunctionType

  FunctionType = SimpleType "⇒" Type | "(" [Types] ")" "⇒" Type

  SimpleType = Byte | Short | Char | Int | Long | Double | Float | Boolean | String

  Types = Type {"," Type}

- Tip Boolean može imati vrednosti *True* i *False*
- Funkcijski tip: Int ⇒ Int, (Int, Int) ⇒ Int, ...

Rezime – elementi jezika

- Definicije:

  Def = FunDef | ValDef
  FunDef = "def" ident ["(" [Params] ")"] [":" Type] "=" Expr
  ValDef = "val" ident [":" Type] "=" Expr
  Param = ident ":" ["⇒"] Type
  Params = Param { "," Param }

- Definicija funkcije: `def square(x: Int) = x*x`
- Definicija vrednosti: `val y = square(2)`

Rezime – elementi jezika

- Izraz može biti:
  - identifikator: `x, isGoodEnough, sqrt, ...`
  - literal: `0, 1.0, "abc", ...`
  - poziv (primena) funkcije: `sqrt(x)`
  - primena operatora: `-x, y + x`
  - selekcija: `Console.println`
  - uslovni izraz: `if(x<0) –x else x`
  - blok: `{ val x = math.abs(y); x*2 }`
  - anonimna funkcija: `x => x+1`