

# Funkcionalno programiranje

## Liste

# Motivacija

- Liste su fundamentalna struktura podataka u jeziku Scala (generalno – u FP)
- Jednostavna rekurzivna definicija
- Homogena struktura podataka
- Za razliku od nizova (Array)
  - Liste su nepromenljive

# Stvaranje liste

```
List('a', 'b', 'c')  
res0: List[Char] = List(a, b, c)
```

```
val fruit = List("apples", "oranges", "pears")  
val nums = List(1, 2, 3, 4)  
val diag3 = List(  
    List(1, 0, 0),  
    List(0, 1, 0),  
    List(0, 0, 1)  
)
```

```
val empty = List()  
empty: List[Nothing] = List()
```

```
val empty2[String] = List()  
empty2: List[String] = List() ←
```

**Nothing** je na dnu hijerarhije klasa

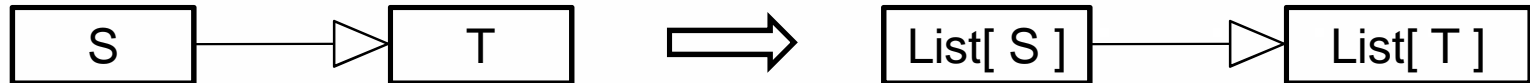
Odnosno – izvodi se iz svih klasa

Ovo je moguće zato što je tip liste u jeziku Scala **kovarijantan**.

# Tip liste je kovarijantan

- Kovarijantnost kolekcije:

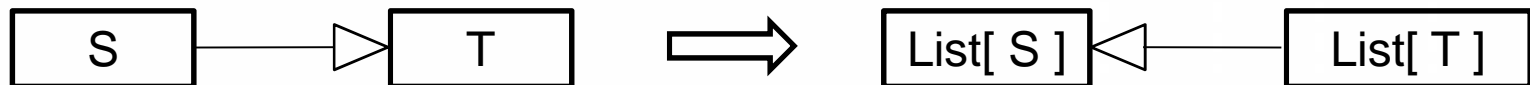
- kolekcija elemenata izvedenog tipa je jedna vrsta kolekcije nadređenog tipa



- kada se otkriva kolekcija nadređenog tipa, može se iskoristiti kolekcija izvedenog tipa

- Kontravarijantnost kolekcije:

- kolekcija elemenata nadređenog tipa je jedna vrsta kolekcije izvedenog tipa



- Podrazumevano, kolekcije su nevarijantne

# Kontravarijantnost

```
class Publication(val title: String)
class Book(title: String) extends Publication(title)
object Library {
  val books: Set[Book] =
    Set( new Book("Programming in Scala"),
          new Book("Walden")
        )
  def printBookList(info: Book => AnyRef) = {
    for (book <- books)      println(info(book))
  }
}

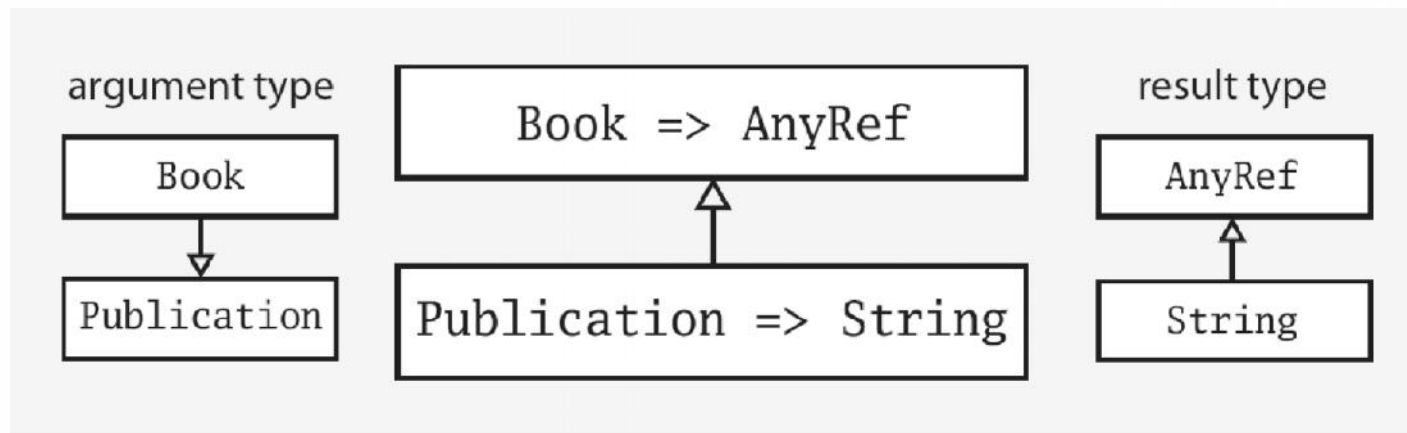
object Customer extends App {
  def getTitle(p: Publication): String = p.title
  Library.printBookList(getTitle)
}
```

# Kontravarijantnost

```
class Publication(val title: String)
class Book(title: String) extends Publication(title)
object Library {
  val books: Set[Book] =
    Set( new Book("Programming in Scala"),
          new Book("Walden")
        )
  def printBookList(info: Book => AnyRef) = {
    for (book <- books)      println(info(book))
  }
}
```

```
trait Function1[-S,+T] {
  def apply(x: S): T
}
```

```
object Customer extends App {
  def getTitle(p: Publication): String = p.title
  Library.printBookList(getTitle)
}
```



# Stvaranje liste (nastavak)

- Sve liste se formiraju na osnovu 2 elementa
  - prazna lista: Nil
  - operacija konstrukcije :: ( ita se – "kons" (*cons*))
- Primer:
  - `x :: xs` pravi novu listu sa prvim elementom x i ostatkom xs
  - `val fruit = "apples"::("oranges"::("pears"::Nil))`
  - `val nums = 1 :: (2 :: (3 :: (4 :: Nil)))`
  - `val empty = Nil`
- Obratiti pažnju
  - zbog smera grupisanja s desna ulevo, `A :: B :: C` interpretira se kao `A :: (B :: C)`
  - dakle, zagrade u prethodnim primerima su bile suvišne
  - `::` je operator desnog operanda: `A :: B` je zapravo `B.::(A)`

# Hijerarhija klase List[T]

- Lista nije primitivan tip u jeziku Scala
  - bazna apstraktna klasa

```
abstract class List[T] {  
  def head : T  
  def tail : List[T]  
  def isEmpty : Boolean  
}
```
  - klasa Nil

```
case object Nil extends List[Nothing]
```
  - klasa ::

```
final case class ::[B](override val head: B,  
  private[scala] var tl: List[B]) extends List[B]
```



# Operacije nad listama

- Sve liste se mogu izraziti u funkciji slede ih operacija
  - head vraća prvi element liste
  - tail vraća naredni element
  - isEmpty vraća true ako je lista prazna

- U klasi List, ove operacije su realizovane kao metode

- Primer:

```
scala> val fruit = "apples"::("oranges"::("pears"::Nil))
fruit: List[String] = List(apples, oranges, pears)
```

```
scala> fruit
```

```
res1: List[String] = List(apples, oranges, pears)
```

```
scala> fruit.head
```

```
res3: String = apples
```

```
scala> fruit.tail
```

```
res2: List[String] = List(oranges, pears)
```

# Primer: insertion sort

```
object isort {
  def insert(i : Int, l : List[Int]) : List[Int] =
  {
    if( l.isEmpty ) List(i)
    else if( i < l.head ) i :: l
    else l.head :: insert(i, l.tail)
  }

  def isort(l : List[Int]) : List[Int] =
  {
    if( l.isEmpty ) Nil
    else insert(l.head, isort(l.tail))
  }

  isort( 3 :: 2 :: 1 :: 5 :: Nil )
  //> res0: List[Int] = List(1, 2, 3, 5)
}
```

# Primer: insertion sort uz upotrebu uparivanja obrazaca

- Pošto su i `Nil` i `::` klase služaja, liste se mogu razložiti primenom uparivanja obrazaca
- Tako `e`, konstruktor `List` se može koristiti kao obrazac, tako što `List(p1, ..., pn)` postaje `p1 :: ... :: pn :: Nil`

```
def isort_pm(l: List[Int]): List[Int] = l match {  
  case List() => List()  
  case h :: t => insert_pm(h, isort_pm(t))  
}  
def insert_pm(e: Int, l: List[Int]): List[Int] = l match {  
  case List() => List(e)  
  case h :: t => if (e <= h) e :: l else h :: insert_pm(e, t)  
}
```

```
isort_pm( 3 :: 2 :: 1 :: 5 :: Nil )  
//> res1: List[Int] = List(1, 2, 3, 5)
```

# Liste i uparivanje obrazaca

- U izrazu, infiksni operator se ponaša kao poziv metode:
  - `h :: t`    `t.::(h)`
- U obrascu se ponaša obrazac konstruktora
  - `h :: t`    `::(t, h)`
- Klasa `::` je pomenuta ranije
  - u paketu `scala` ( `scala.::` )
  - pravi neprazne liste
- Dakle ime `::` postoji na dva mesta u jeziku Scala
  - kao klasa
  - kao metoda klase `List`, koja pravi instancu klase `scala.::`

# Primer: funkcija za određivanje dužine liste

```
def length[A](l: List[A]): Int = l match
{
  case List() => 0
  case h :: t => 1 + length(t)
}
```

```
length( 3 :: 2 :: 1 :: 5 :: Nil)
//> res2: Int = 4
```

# Druge operacije nad listama

## Poglavlje 16.6

- Operator `::` je asimetričan
  - primenjuje se nad podatkom i listom
- Postoji poseban operator konkatencije `:::` (concat)
  - spaja dve liste i formira novu
- `last`
  - dohvata poslednji element liste (suprotno od `head`)
- `init`
  - dohvata sve elemente liste sem poslednjeg (suprotno od `tail`)
- `reverse`
  - vraća listu u obrnutom poretku elemenata

# Druge operacije nad listama

```
def concat[A](l1 : List[A], l2 : List[A]) : List[A] = l1 match {  
  case List() => l2  
  case h :: t => h :: concat(t, l2)  
}
```

```
val a = 1 :: 2 :: 3 :: Nil  
//> a : List[Int] = List(1, 2, 3)
```

```
concat(a, 5 :: 1 :: 0 :: Nil)  
//> res3: List[Int] = List(1, 2, 3, 5, 1, 0)
```

```
def last[A](l : List[A]) : A = l match {  
  case List() => error("Empty list")  
  case h :: Nil => h // ili List(x) => x  
  case h :: t => last(t)  
}
```

```
def error(msg: String): Nothing = throw new  
  RuntimeException(msg)
```

```
last(a) //> res4: Int = 3
```





# Metode višeg reda klase List

Poglavlje 16.7

- map – formira novu listu pretvaraju i sve elemente date liste na osnovu funkcije  $T \Rightarrow U$

primeri:

```
List(1,2,3) map (_ + 1) // map (x=>x+1)
```

```
val words = List("the", "quick", "brown", "fox")  
words map (_.toList.reverse.mkString)  
// List[String] = List(eht, kciuq, nworb, xof)
```

# Metode višeg reda klase List

- flatMap – formira listu spajanjem sadržaja podlista  
primer:

```
words map (_.toList)
```

```
// List[List[Char]] = List(List(t, h, e),  
List(q, u, i, c, k), List(b, r, o, w, n),  
List(f, o, x))
```

```
words flatMap (_.toList)
```

```
// List[Char] = List(t, h, e, q, u, i, c, k, b,  
r, o, w, n, f, o, x)
```

# Metode višeg reda klase List

- `filter` – formira listu izostavljanjem elemenata koji ne zadovoljavaju dati uslov

```
List(1, 2, 3, 4, 5) filter (_ % 2 == 0)  
// List[Int] = List(2, 4)
```

```
words filter (_.length == 3)  
// List[java.lang.String] = List(the, fox)
```

- `partition` – formira dve liste na osnovu datog uslova

```
List(1, 2, 3, 4, 5) partition (_ % 2 == 0)  
// res42: (List[Int], List[Int]) =  
          (List(2, 4), List(1, 3, 5))
```

# Metode višeg reda klase List

- `find` – pronalazi i vraća **prvi** element koji zadovoljava uslov. Vraća `None` ako nijedan element ne zadovoljava uslov.

```
List(1, 2, 3, 4, 5) find (_ % 2 == 0)  
// res43: Option[Int] = Some(2)
```

```
List(1, 2, 3, 4, 5) find (_ <= 0)  
// res44: Option[Int] = None
```

# Metode višeg reda klase List

- `takeWhile` – formira novu listu koja sadrži najduži prefiks date liste kod kojeg svi elementi zadovoljavaju uslov

```
List(1, 2, 3, -4, 5) takeWhile (_ > 0)
// res45: List[Int] = List(1, 2, 3)
```

- `dropWhile` – formira novu listu koja ne sadrži najduži prefiks date liste kod kojeg svi elementi zadovoljavaju uslov

```
words dropWhile (_ startsWith "t")
// res46: List[java.lang.String] =
      List(quick, brown, fox)
```

# Metode višeg reda klase List

- `span` – formira nov par lista, kod kojih prvu listu najduži prefiks date liste definisan predikatom `p`, a drugu ostatak.

`xs span p`  $\leftrightarrow$  `(xs takeWhile p, xs dropWhile p)`

- Izbegava dvostruki prolaz kroz listu

# Metode višeg reda klase List

- `forall` – vraća `true` ako svi elementi liste zadovoljavaju zadati uslov.
- `exists` – vraća `true` ako najmanje jedan element liste zadovoljava zadati uslov.

```
val diag3: List[List[Int]] =  
  List( List(1, 0, 0), List(0, 1, 0),  
        List(0, 0, 1) )
```

```
def hasZeroRow(m: List[List[Int]]) =  
  m exists (row => row forall (_ == 0))
```

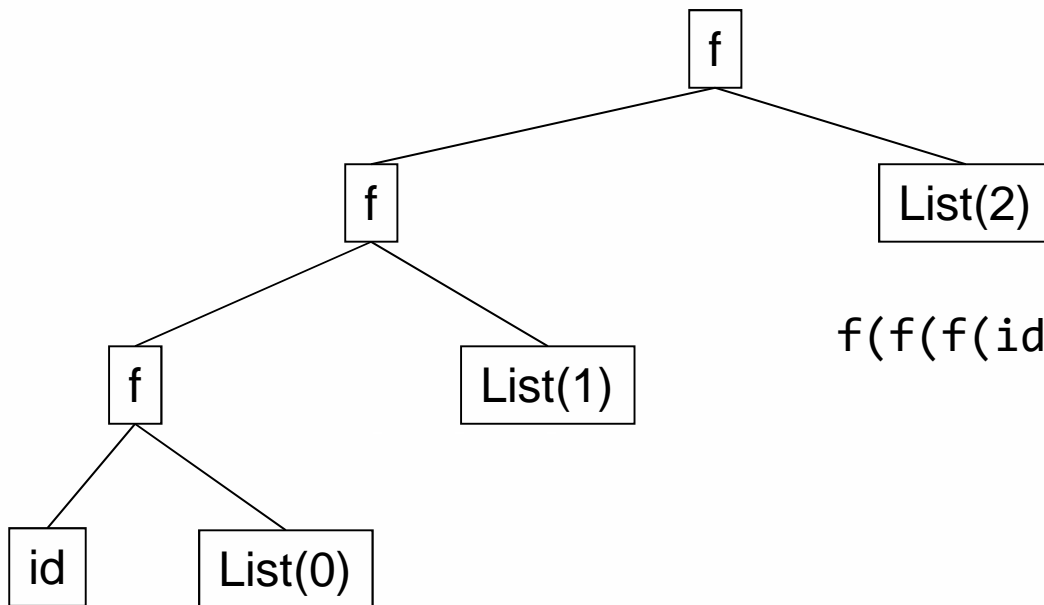
```
scala> hasZeroRow(diag3)  
res48: Boolean = false
```

# Metode višeg reda klase List

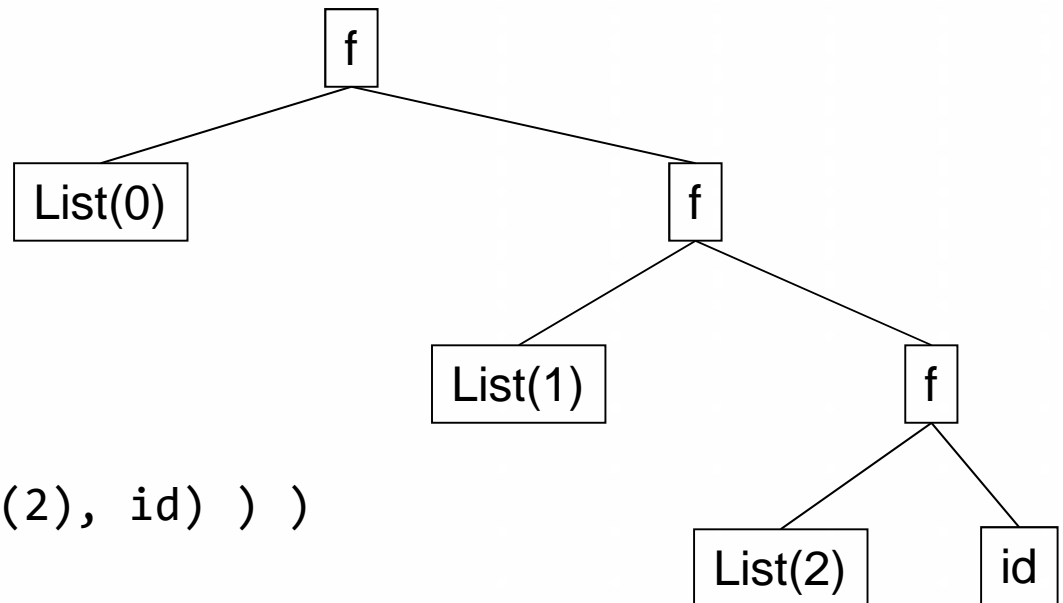
- Odmotavanje (eng. folding) liste: operatori  $/:$  i  $:\backslash$
- Ovi operatori:
  - primaju po jednu vrednost  $id$  i listu
  - vraćaju funkciju :  $((T, T) \Rightarrow T) \Rightarrow T$  Ovo je funkcija  $f$ .
- Suštinski:  $(id /: list)(f)$ ,  
odnosno  $(list :\backslash id)(f)$
- Funkcija  $f$  se primenjuje na sve elemente liste
- Operator levog odmotavanja  $/:$ 
  - Prvo se primenjuje  $f(id, list(0))$  /\* = tmp \*/  
Zatim  $f(tmp, list(1))$ , itd
  - $f(f(f(id, list(0)), list(1)), list(2))$
- Operator desnog odmotavanja  $:\backslash$ 
  - $f(list(0), f(list(1), f(list(2), id) ) )$



# Metode višeg reda klase List



`f(f(f(id, list(0)), list(1)), list(2))`



`f(list(0), f(list(1), f(list(2), id) ) )`

# Metode višeg reda klase List

- Primeri *levog* odmotavanja:
- `def sum(l: List[Int]): Int = (0 /: l) (_ + _)`  
Podsetnik: `(_ + _)` je skra eno `( (x,y)=>x+y )`
- `scala> sum( List(1,2,3) )`  
`Res8: Int = 6`
- `def prod(l: List[Int]): Int = (1 /: l) (_ * _)`
- `scala> prod( List(1,2,3,4) )`  
`Res9: Int = 24`

# Metode višeg reda klase List

- Da li je uvek sve jedno da li se koristi /: ili :\ ?

```
def flattenLeft[T](xss: List[List[T]]) =  
  (List[T]() /: xss) (_ ::: _)
```

```
def flattenRight[T](xss: List[List[T]]) =  
  (xss :\ List[T]()) (_ ::: _)
```

- flattenRight je generalno prirodniiji, jer levi operand operatora ::: je uvek pojedinačan element (koga god tipa bilo), a ne lista svih elemenata koji su prethodno obrađeni.
- U konkretnom slučaju, flattenRight je efikasniji