Funkcionalno programiranje

Vežbe

04 Uparivanje obrazaca

Zadatak 4.1

- Napisati funkciju čija vrednost je deskriptivni opis (string) njenog argumenta upotrebom mehanizma uparivanja obrazaca.

```
package w05
object patternmatchingexamples {

  class Rational(x: Int, y: Int) {
    require(y != 0)
    private def gcd(a: Int, b: Int): Int = if(b==0) a else gcd(b, a%b)
    private val g = gcd(x.abs, y.abs)
    def numer = x / g
    def denom = y / g
    override def toString() = numer + "/" + denom
  }

  case class cRational(x : Int, y : Int) extends Rational(x, y)
  val cc = cRational(2,4)
  //> cc  : w05.patternmatchingexamples.cRational = 1/2
```

Zadatak 4.1

```
type F = (Int => Int)

def opisTipa(x: Any): String = x match {

  // Obrazac konstante
  case 0 => "nula"
  case true => "tacno"
  case "zdravo" => "tekst poruka 'zdravo'"
  case Nil => "Prazna lista"

  // Obrazac sekvence
  case List(0, _, _) => "Lista od 3 elementa koja pocinje nulom"
  case List(1, _*) =>
     "Lista proizvoljnog broja elemenata koja pocinje jedinicom"
  case List(d : Double, _*) =>
     "Lista proizvoljnog broja elemenata koja pocinje sa Double"
  case Vector(1, _*) =>
    "Vektor proizvoljnog broja elemenata koji pocinje jedinicom"
```

Zadatak 4.1

```
// Obrazac taplova
case (a, b) => s"Par sa elementima: $a i $b"
case (a, b, c) => s"Trojka sa elementima: $a, $b i $c"

// Obrasci konstruktora
case cRational(brojilac, 5) =>
    s"Racionalan broj, sa imeniocom 5 i brojiocem = $brojilac"

case cRational(brojilac, imenilac) =>
    s"Racionalan broj, $imenilac / $brojilac"
```

Zadatak 4.1

```
  // Obrasci tipa
  case s: String => s"String: $s"
  case i: Int => s"Ceo broj: $i"
  case f: Float => s"Realan broj: $f"
  case a: Array[Int] => s"Niz celih brojeva: ${a.mkString(",")}"
  case as: Array[String] => s"Niz stringova: ${as.mkString(",")}"
  case d: cRational => s"Racionalan broj: ${d}"
  case list: List[_] => s"Lista bilo kog tipa: $list"

  case m: Map[_, _] => m.toString

  case f : (Int => Int) => "Funkcija Int => Int"
  case f1 : F => "Funkcija Int => Int"

  // Obrazac koji uparuje sve
  case _ => "Nepoznato"

} //> opisTipa: (x: Any)String
```

non-variable type argument Int in type pattern Int ⇒ Int is unchecked since it is eliminated by erasure

Zadatak 4.1

```
  opisTipa( Array(1,2,3,4) )
  //> res0: String = Niz celih brojeva: 1,2,3,4
  opisTipa( Array("s1", "s2", "s3"))
  //> res1: String = Niz stringova: s1,s2,s3
  opisTipa( Array(1.0,2.0,3.0,4.0) )
  //> res2: String = Nepoznato

  opisTipa( List(1.0, 2, List('a','b')) )
  //> res3: String = Lista proizvoljnog broja elemenata koja pocinje
  //| jedinicom

 opisTipa( List(3.0, 2, List('a','b')) )
  //> res6: String = Lista proizvoljnog broja elemenata koja pocinje
  //| sa Double

  opisTipa( (x : Int ) => x)
  //> res4: String = Funkcija Int => Int
  opisTipa( (y : Float) => y*2 )
  //> res5: String = Funkcija Int => Int
}
```

Zadatak 4.1

```
case List(1, _*) =>
      "Lista proizvoljnog broja elemenata koja pocinje jedinicom"

case list: List[_] => s"Lista bilo kog tipa: $list"

case list2 : List(2, _*) =>
      s"Lista koja pocinje dvojkom: $list2"        // greška!
```

Da bi se ovo realizovalo, potrebno je kombinovati
obrazac vezivanja promenljive za obrazac sekvence

```
case list2 @ List(2, _*) =>
      s"Lista koja pocinje dvojkom: $list2"        // ispravno
```

Zadatak 4.2

- (a) Napisati funkciju za nalaženje najvećeg zajedničkog delioca dva cela broja primenom uparivanja obrazaca.

```
object gcd {
  def gcd(first: Int, second: Int): Int = (first, second) match {
    case (0, b) => b
    case (a, 0) => a
    case (a, b) if a > b => gcd(a - b, b)
    case (a, b) => gcd(a, b - a)
  } //> gcd: (first: Int, second: Int)Int

  gcd(3, 5)                                       //> res0: Int = 1
}
```

Zadatak 4.2

- (b) Proširiti funkcionalnost klase String tako da može da konvertuje znakovni niz u logičku vrednost:
  - `"", " ", "0", "zero"` u `false`
  - bilo koja druge vrednost u `true`

```
  implicit class StringImprovements(val s: String) {

    def asBoolean = s match {
      case "0" | "zero" | "" | " " => false
      case _ => true
    }
}

"zero".asBoolean                          //> res0: Boolean = false
"0".asBoolean                             //> res1: Boolean = false

"test".asBoolean                          //> res2: Boolean = true
if( "test".asBoolean ) true else false    //> res3: Boolean = true
```

Zadatak 4.2

- Komentar: funkcionalnost klase String je proširena, ali neće biti implicitno korišćena.

- Za implicitnu upotrebu treba napisati implicitne funkcije:

```
implicit def strToBool(s: String) : Boolean = s match {
  case "0" | "zero" | "" | " " => false
  case _ => true
}

if( "test" ) true else false  //> res3: Boolean = true
```

Zadatak 4.2

- (c) Proširiti funkcionalnost klase `String` dodavanjem bezbedne konverzije iz tipa `String` u tip `Int`, bez bacanja izuzetaka.

```
  implicit class StringImprovements(val s: String) {

    def toInteger : Option[Int] = {
      try {
        Some(Integer.parseInt(s))
      } catch {
        case e: NumberFormatException => None
      }
    }
}

"test".toInteger match {
    case Some(x) => println(x)
    case None => println("Not a number")
}                                          //> Not a number
```

Zadatak 4.3

- Novčanik sadrži određen broj kovanica u apoenima 1..5 dinara i novčanica u apoenima 10..5000 dinara. Sadržaj novčanika je predstavljen listom parova (apoen, količina). Odrediti:
  - da li je novčanik prazan?
  - da li ima novčanice?
  - koliko novčanica apoena 200 dinara sadrži?

Zadatak 4.3

```
val novcanik = List( (10, 0), (20, 1), (50, 2), (100, 5), (200, 3),
                     (500, 2), (1000, 1), (2000, 0), (5000, 0) )

novcanik match {

  case l : List[_] if l.length < 1 => "Prazan"

  case l : List[(Int,Int)] // ili List[(_,_)]
     if l.forall(x => x._2 == 0) => "Nema novcanica od 10 do 5000"


  case l : List[(Int,Int)]
     if l.exists(x => (x._1 == 200 && x._2 > 0)) =>
       s"Sadrzi ${l.find(x => x._1 == 200).get._2} novcanica po 200"

}
//> res6: String = Sadrzi 3 novcanica po 200
```

Zadatak 4.4

- Napisati potrebne klase za predstavljanje i računanje vrednosti numeričkih izraza i obezbediti njihov ispis.
  - Obezbediti operacije +, - i *
  - Pri ispisu koristiti zagrade samo kada su neophodne

Zadatak 4.4
Rešenje 1

```
object expression {

  trait Expression
  case class Number(n: Int) extends Expression
  case class Variable(s : String, var n : Int) extends Expression
  case class Add(e1: Expression, e2: Expression) extends Expression
  case class Sub(e1: Expression, e2: Expression) extends Expression
  case class Prod(e1: Expression, e2: Expression) extends Expression


  val c = Number(2)
                //> c  : w05.expression.Number = Number(2)
  println(c)
                //> Number(2)
```

Zadatak 4.4
Rešenje 1

```
def eval(e: Expression): Int = e match {
  case Number(n) => n
  case Variable(_, n) => n
  case Add(e1, e2) => eval(e1)+eval(e2)
  case Sub(e1, e2) => eval(e1)-eval(e2)
  case Prod(e1, e2) => eval(e1)*eval(e2)
  case _ => 0
}

eval(Sub(Number(3), Number(2)))           //> res0: Int = 1
eval(Sub(Number(4), Number(2)))           //> res1: Int = 2
eval(Prod(Number(2), Variable("x", 3)))   //> res2: Int = 6

val a = Variable("a", 5)
                  //> a  : w05.expression.Variable = Variable(a,5)
eval(a)                                   //> res3: Int = 5
a.n = 6
eval(a)                                   //> res4: Int = 6
```

Zadatak 4.4
Rešenje 1

```
def show(e: Expression): String = e match {
  case Number(n) => ""+n
  case Variable(s, _) => s

  case Prod(Add(e1, e2), e3) =>
         "(" + show(e1) + "+" + show(e2) + ")" + "*" + show(e3)

  case Prod(Sub(e1, e2), e3) =>
         "(" + show(e1) + "-" + show(e2) + ")" + "*" + show(e3)

  case Add(e1, e2) => show(e1) + "+" + show(e2)
  case Sub(e1, e2) => show(e1) + "-" + show(e2)
  case Prod(e1, e2) => show(e1) + "*" + show(e2)
  case _ => "???"
}
```

Zadatak 4.4
Rešenje 1

```
show(Number(3))                                  //> res5: String = 3

show(Add(Number(2), Number(3)))                  //> res6: String = 2+3

show(Add(Number(3), Sub(Number(4), Number(2))))
                                                 //> res7: String = 3+4-2

show(Prod(Number(2), Variable("x", 3)))          //> res8: String = 2*x

show(Add(Prod(Number(2), Number(3)), Variable("x", 4)))
                                                 //> res9: String = 2*3+x

show(Prod(Add(Number(2), Number(3)), Variable("x", 4)))
                                                 //> res10: String = (2+3)*4

show(Prod(Variable("y", 5), Sub(Variable("x", 3), Number(2))))
                                                 //> res11: String = y*x-2
```

Zadatak 4.4
Rešenje 1

- Analiza rešenja
- Posebno moraju da se razmatraju slučajevi
  - x * (y+z)
  - (y+z) * x
- Posebno se razmatra
  - zbir od proizvoda zbirovaa
  - razlika od proizvoda razlika
- Mnogo slučajeva koje je potrebno pokriti
- Nepraktično!

Zadatak 4.4
Rešenje 2

```
object expression2 {

  trait Expr {

    def +(that: Expr): Expr = (this, that) match {

      case (Num(0), e) => e

      case (Num(n), Num(m)) => Num(n+m)

      case (Var(x, a), Var(y, b)) if x==y => Num(2) * Var(x, a)

      case (Mul(e1, Var(x, a)), Mul(e2, Var(y, b))) if x==y =>
              (e1 + e2) * Var(x, a)

      case _ => Add(this, that)
    }
```

Zadatak 4.4
Rešenje 2

```
def -(that: Expr): Expr = (this, that) match {
  case (Num(0), e) => Mul(Num(-1), e)
  case (e, Num(0)) => e
  case (Num(n), Num(m)) => Num(n-m)
  case (Var(x, _), Var(y, _)) if x==y => Num(0)

  case (Mul(e1, Var(x, a)), Mul(e2, Var(y, b))) if x==y =>
            (e1 - e2) * Var(x, a)

  case _ => Sub(this, that)
}

def *(that: Expr): Expr = (this, that) match {
  case (Num(0), e) => Num(0)
  case (Num(1), e) => e
  case (Num(n), Num(m)) => Num(n*m)
  case (Var(x, a), Num(n)) => Num(n) * Var(x, a)
  case _ => Mul(this, that)
}
```

Zadatak 4.4
Rešenje 2

```
  def unary_! : Int = this match {
    case Num(x) => x
    case Var(_, x) => x
    case Add(a, b) => !a + !b
    case Sub(a, b) => !a - !b
    case Mul(a, b) => !a * !b
  }
}

case class Num(x: Int) extends Expr {
       override def toString = x.toString
}

case class Var(name: String, x : Int) extends Expr {
       override def toString = name
}
```

Zadatak 4.4
Rešenje 2

```
case class Add(e1: Expr, e2: Expr) extends Expr {
  override def toString = e1.toString + " + " + e2.toString
}

case class Sub(e1: Expr, e2: Expr) extends Expr {
  override def toString = e1.toString + " - " + e2.toString
}

case class Mul(e1: Expr, e2: Expr) extends Expr {
  override def toString = {
    def factorToString(e: Expr) = e match {
      case Add(_, _) => "(" + e.toString + ")"
      case Sub(_, _) => "(" + e.toString + ")"
      case _ => e.toString
    }

    factorToString(e1) + " * " + factorToString(e2)
  }
}
```

Zadatak 4.4
Rešenje 2

```
  !Num(3)                                          //> res0: Int = 3
  !(Num(2) + Num(3))                               //> res1: Int = 5
  !(Num(3) + Num(4) - Num(2))                      //> res2: Int = 5
  !(Num(2) * Var("x", 3))                          //> res3: Int = 6

  val r4= (Num(2) * Num(3) + Var("x", 4))
  println(r4)                                      //> 6 + x
  !r4                                              //> res4: Int = 10

  val r5 = (Num(2) * (Num(3) + Var("x", 4)))
  println(r5)                                      //> 2 * (3 + x)
  !r5                                              //> res5: Int = 14

  val r6 = (Var("y", 5) * (Var("x", 3) - Num(2)))

  println(r6)                                      //> y * (x - 2)
  !r6                                              //> res6: Int = 5
}
```

Zadatak 4.5

- Napisati potrebne klase za predstavljanje i manipulaciju stablima binarnog pretraživanja sa proizvoljnim tipom sadržaja čvorova uz upotrebu tehnike uparivanja obrazaca.

Zadatak 4.5
Rešenje 1 – samo za tip Int

```
object tree {

  abstract class IntTree

  case class Empty() extends IntTree

  case class Node(elem: Int, left: IntTree, right: IntTree)
    extends IntTree

  object Node {
    // Redundantno, kompajler prijavljuje gresku
    // def apply(elem: Int, left: IntTree, right: IntTree) =
    //     new Node(elem, left, right)
    def apply(elem: Int) = new Node(elem, Empty(), Empty())
  }
```

Zadatak 4.5
Rešenje 1

```
def contains(t: IntTree, v: Int): Boolean = t match {
  case n : Empty => false
  case Node(el, _, _) if(el == v) => true
  case Node(el, left, right) =>
    if( v < el ) contains(left, v) else contains(right, v)
}

def add(t: IntTree, v: Int) : IntTree = t match {
  case n : Empty => Node(v, Empty(), Empty())
  case Node(el, _, _) if( el == v ) => t
  case Node(el, left, right) =>
    if( v < el ) Node(el, add(left, v), right)
    else Node(el, left, add(right, v))
}
```

Zadatak 4.5
Rešenje 1

```
  val root  = Node(10, Node(3), Empty())
  //> root  : w05.tree.Node = Node(10,Node(3,Empty(),Empty()),Empty())

  contains(root, 5)
  //> res0: Boolean = false

  contains(root, 10)
  //> res1: Boolean = true

  contains(root, 3)
  //> res2: Boolean = true

  add(root, 5)
  //> res3: w05.tree.IntTree =
  //| Node(10,Node(3,Empty(),Node(5,Empty(),Empty())),Empty())
}
```

Zadatak 4.5
Rešenje 2

```
object tree2 {

  abstract class Tree[T]

  case class Empty[T]() extends Tree[T]

  case class Node[T](elem: T, left: Tree[T], right: Tree[T])
      extends Tree[T]

  object Node {
    def apply[T](elem: T) = new Node(elem, Empty(), Empty())
  }
```

Zadatak 4.5
Rešenje 2

```
def contains[T](t: Tree[T], v: T)
                (implicit ord: Ordering[T]): Boolean = t match {
  case n : Empty[T] => false
  case Node(el, _, _) if(el == v) => true
  case Node(el, left, right) =>
    if( ord.gt(el, v) ) contains(left, v)
    else contains(right, v)
}

def add[T](t: Tree[T], v: T)
          (implicit ord: Ordering[T]) : Tree[T] = t match {
  case n : Empty[T] => Node(v, Empty(), Empty())
  case Node(el, _, _) if( el == v ) => t
  case Node(el, left, right) =>
    if( ord.gt(el, v) ) Node(el, add(left, v), right)
    else Node(el, left, add(right, v))
}
```

Zadatak 4.5
Rešenje 2

```
  val root  = Node(10, Node(3), Empty())
  contains(root, 5)                          //> res0: Boolean = false
  contains(root, 10)                         //> res1: Boolean = true
  contains(root, 3)                          //> res2: Boolean = true

  add(root, 5)
  //> res3: w05.tree2.Tree[Int] =
  //| Node(10,Node(3,Empty(),Node(5,Empty(),Empty())) ,Empty())

  var root2 = Node("ABC")
  root2 = add(root2, "ADE")
  root2 = add(root2, "XYZ")
  root2 = add(root2, "ADA")
  //> root2  : w05.tree2.Tree[String] =
  //| Node(ABC,Empty(),Node(ADE,Node(ADA,Empty(
  //| ),Empty()),Node(XYZ,Empty(),Empty())))
}
```

Zadatak 4.5

Rešenje 3 – kombinovanje sa OO dekompozicijom

```
object tree3 {

  abstract class Tree[T] {
    def contains(v: T)
                (implicit ord: Ordering[T]): Boolean = this match {
      case n: Empty[T] => false
      case Node(el, _, _) if(el == v) => true
      case Node(el, left, right) =>
        if( ord.gt(el, v) ) left contains el
        else right contains el
    }

    private[tree3] def addInternal(v: T)(ord: Ordering[T]): Tree[T]

    def add(v: T)(implicit ord: Ordering[T]): Tree[T] = {
      if( this contains v ) this
      else addInternal(v)(ord)
    }
```

Zadatak 4.5

Rešenje 3 – kombinovanje sa OO dekompozicijom

```
  override def toString = this match {
    case n : Empty[T] => "."
    case Node(el, l, r) =>
      "(" + (l toString) + "_" + el + "_" + (r toString) + ")"
  }
}

case class Empty[T]() extends Tree[T] {
    override def addInternal(v: T)
                              (ord: Ordering[T]): Tree[T] = Node(v)
}

case class Node[T](elem: T, left: Tree[T], right: Tree[T])
     extends Tree[T] {
  override def addInternal(v: T)(ord: Ordering[T]): Tree[T] = {
    if( ord.gt(elem, v) )
      Node(elem, left.addInternal(v)(ord), right)
    else Node(elem, left, right.addInternal(v)(ord))
  }
}
```

Zadatak 4.5

Rešenje 3 – kombinovanje sa OO dekompozicijom

```
    object Node {
      def apply[T](elem: T) = new Node(elem, Empty(), Empty())
    }

    val root = Empty().add(10).add(5).add(15)
    //> root  : w05.tree3.Tree[Int] = ((._5_.)_10_(._15_.))

    root contains 3                          //> res0: Boolean = false

    val root2 = Node("ABC").add("ADE").add("XYZ").add("ADA")
    //> root2  : w05.tree3.Tree[String] =
    //| (._ABC_((._ADA_.)_ADE_(._XYZ_.)))
}
```