Funkcionalno programiranje

Uparivanje obrazaca (pattern matching)

Motivacija

- Proširivanje hijerarhije klasa
  - dodavanje nove metode – potencijalno zahteva implementaciju u svim potklasama
  - proširivanje funkcionalnosti dodavanjem nove kategorije – zahteva metode za određivanje pripadnosti datoj kategoriji (napr: potrebno utvrditi parametre konstruktora)
  - primer: numerički izrazi i naknadno dodavanje ispisa
- Moguće je rešiti u osnovnoj klasi
  - eksplicitnim utvrđivanjem tipa i konverzijom
  - sklono greškama

Klase slučaja (case classes)

- Definišu se kao "obične" klase
- Prethodi ključna reč `case`

U `case` klasama se podrazumeva `val`

`case` klase se ne mogu izvesti iz drugih `case` klasa

```
trait Expression
case class Number(n: Int) extends Expression
case class Sum(e1: Expression, e2: Expression) extends Expression
case class X extends Number(5)        // Ne može
class Y extends Number(5)             // Može
case class Z extends Y                // Ne može
```

- Implicitno se definišu prateći objekti sa apply metodama

```
object Number { def apply(n: Int) = new Number(n) }
object Sum { def apply(e1: Expression, e2:Expression) =
        new Sum(e1, e2) }
```

To su proizvodne metode (factory method): dovoljno `Number(1)` umesto `new Number(1)`

Klase slučaja (case classes)

- Implicitno se definišu "prirodne" implementacije:
  - `toString`
  - `hashCode`
  - `equals`
- Implicitno se generiše `copy` metoda
  - omogućava pravljenje nove instance na osnovu postojeće, korisno kada se razlikuju u nekoliko atributa
  - koriste se imenovani parametri

```
scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))

scala> op.copy(operator = "-")
res4: BinOp = BinOp(-,Number(1.0),Var(x))
```

Postoji atribut klase: "operator"

Klase slučaja (case classes)

- U prethodnom primeru nisu definisane metode klasa
- Razlog: klase će biti korišćene za uparivanje obrazaca

```
def eval(e: Expression): Int = e match
   {
        case Number(n) => n
        case Sum(e1, e2) => eval(e1) + eval(e2)
   }
```

e se naziva **selektor**

Primetiti: ne koristi se polimorfizam

Sintaksa:

```
e match {
   case obrazac => izraz
   ...
}
```

Uparivanje se vrši u redosledu navođenja

Baca se izuzetak `MatchError`
ako nijedan obrazac ne bude uparen

Uparivanje obrasca kao metoda klase

```
abstract class Expr
{
   def eval: Int = this match
   {
        case Number(n) ⇒ n
        case Sum(e1, e2) ⇒ e1.eval + e2.eval
   }
}
```

- Primetiti: ovo je metoda nadklase klasa Number i Sum
- Šta je bolje? OO dekompozicija ili uparivanje obrazaca?
  - ako je potrebna nova klasa, onda OO dekompozicija
  - ako je potrebna nova metoda (propagira se u sve potklase), onda uparivanje obrazaca

Šta može biti obrazac?

promenljive `n, e1, e2`	identifikator mora početi malim slovom
konstante `5, true`, ...	identifikator mora početi VELIKIM slovom izuzev u slučaju rezervisanih reči (napr. true)
tipovi `s` : String	uparuje podatak datog tipa
konstruktori `Number, Sum`	promenljivama se naznačavaju evenutalni argumenti konstruktora: `Number(n)`
džoker znak `_`	kada nije bitan argument: `Number(_)`

Ime promenljive se može pojaviti **samo jednom**

- `Sum(e1, e1)` nije dozvoljen obrazac

Šta može biti obrazac?

- Primer upotrebe džoker znaka:

```
expr match {

    case BinOp(_, _, _) => // BinOp(op, left, right)
           println(expr +" je binarna operacija")


    case _ =>
}
```

Šta može biti obrazac?

- Konstruktorski obrazac za klasu `UnOp`: `UnOp("-", e)`
  - uparuje sve vrednosti tipa `UnOp` čiji prvi argument je jednak "-", a drugi uparuje `e`.
- Obrasci se mogu kombinovati iz osnovnih oblika
  - argumenti konstruktora su takođe obrasci
  - "duboka uparivanja" (za strukture poput stabala)
- Primer:

```
expr match {

    case Sum(Number(1), x) => 1 + eval(x)

    case UnOp("-", UnOp("-", e)) => e
}
```

Šta može biti obrazac?

- Obrasci mogu biti linearni tipovi, poput `List` ili `Array`
- Sintaksa je ista
- Moguće je navesti proizvoljan broj elemenata u obrascu

```
expr match {
   case List(0, _, _) => println("Lista")
   case _ =>
}
```

```
expr match {
   case List(0, _*) => println("Lista 2")
   case _ =>
}
```

Šta može biti obrazac?

- Obrasci mogu biti **taplovi**

```
def tupleDemo(expr: Any) = expr match {
   case (a, b, c) => println("matched "+ a + b + c)
   case _ =>
}

tupleDemo: (expr: Any)Unit

scala> tupleDemo(("jedan ","troelementni ","tapl"))
matched jedan troelementni tapl
```

Šta uparuje obrazac?

- Obrazac konstruktora **C(p_1, ..., p_n)** uparuje sve vrednosti tipa **C** (ili podtipa **C** – podtip nije case klasa) napravljene sa argumentima koje uparuju obrasci **p_1...p_n**
- Obrazac promenljive **x** uparuje **bilo koju vrednost** i vezuje ime promenljive za tu vrednost
- Obrazac konstante **c** uparuje vrednosti == **c**

Postupak uparivanja

```
def eval(e:Expression):Int = e match {
  case Number(n) ⇒ n
  case Sum(e1, e2) ⇒ e1.eval + e2.eval
 }
}
```

```
   eval(Sum(Number(1), Number(2)))
→  Sum(Number(1), Number(2)) match {
       case Number(n) ⇒ n
       case Sum(e1, e2) ⇒ eval(e1) + eval(e2)
   }
→ eval(Number(1)) + eval(Number(2))
→ Number(1) match {
       case Number(n) ⇒ n
       case Sum(e1, e2) ⇒ eval(e1) + eval(e2)
   } + eval(Number(2))
→  1 + eval(Number(2))
→  1 + 2 → 3
```

Primetiti: referisane promenljive u obrascu zamenjuju se parametrima konstruktora:

Number(1) ⟶ Number(n), n=1

Obrasci na osnovu tipa

- Veoma praktičan način za utvrđivanje ili konverziju tipa
- Primer

```
def generalSize(x: Any) = x match {
   case s: String => s.length
   case m: Map[_, _] => m.size
   case _ => -1
}
```

Uparuje svaku instancu tipa String (**x ne može biti null**)

```
scala> generalSize("abc")
res16: Int = 3

scala> generalSize(Map(1 -> 'a', 2 -> 'b'))
res17: Int = 2

scala> generalSize(math.Pi)
res18: Int = -1
```

Obrasci na osnovu tipa i brisanje tipa

- Scala koristi mehanizam brisanja tipa (type erasure) kao i Java
- To znači da se u vreme izvršenja programa **ne čuva** informacija o tipu generičkog parametra
- Primer:

```
def isIntIntMap(x: Any) = x match {
   case m: Map[Int, Int] => true
   case _ => false
}

scala> isIntIntMap(Map(1 -> 1))
res19: Boolean = true

scala> isIntIntMap(Map("abc" -> "abc"))
res20: Boolean = true
```

Od ovog pravila odstupa Array : tip elemenata **pamti se** u samom nizu

Vezivanje promenljive

- Moguće je vezati promenljivu za deo obrasca (koji je takođe – obrazac)
- Smisao: obaviti uparivanje na uobičajen način
  - ako uparivanje uspe, veži promenljivu za uparen objekat
- Sintaksa: *ime_promenljive @ obrazac*
- Primer: unarna operacija "abs" se primenjuje 2 puta
  - dovoljno je zadržati jednu primenu

```
expr match {
   case UnOp("abs", e @ UnOp("abs", _)) => e
   case _ =>
}
expr match { //ne može, zato što meša uparivanje tipa i konstruktora
   case UnOp("abs", e) => e match {
      case op : UnOp("abs", _ ) => op
   }
}
```

Provera obrasca (pattern guard)

- Postoje situacije kada sintaksno uparivanje nije dovoljno precizno
  - na primer: potrebno je utvrditi da su dva parametra obrasca identična

```
BinOp("+", Var("x"), Var("x"))  →  BinOp("*", Var("x"), Number("2"))
```

```
scala> def simplifyAdd(e: Expr) = e match {
   case BinOp("+", x, x) => BinOp("*", x, Number(2))
   case _ => e
}
<console>:11: error: x is already defined as value x case
BinOp("+", x, x) => BinOp("*", x, Number(2))
```

Provera obrasca (pattern guard)

- Provera obrasca omogućava uspešno uparivanje samo ako je logički uslov ispunjen

```
scala> def simplifyAdd(e: Expr) = e match {
   case BinOp("+", x, y) if x == y => BinOp("*", x, Number(2))
   case _ => e
}
```

- Provera može biti bilo kakav uslov
  - obično je vezan za promenljive u obrascu
  - ali nije neophodno

```
// Pozitivni celi brojevi
case n: Int if 0 < n => ...

// String koji pocinje slovom ‘a’
case s: String if s(0) == 'a' => ...
```

Uparivanje obrazaca i zapečaćene (sealed) klase

- Scala ima drugačije tumačenje zapečaćene klase u odnosu na C#
  - sealed: klasa može biti nasleđena, ali klase-naslednice **moraju biti definisane u istom fajlu** kao i roditeljska
  - naslednici klasa-naslednica mogu biti i u drugim fajlovima

- Zapečaćene klase su korisne kod uparivanja obrazaca
  - kompajler može da prepozna da li su sve mogućnosti uparivanja pokrivene
  - izbacuje upozorenje na moguć MatchError izuzetak
  - anotacija `@unchecked` omogućava da kompajler ne upozorava

Uparivanje obrazaca i zapečaćene (sealed) klase

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class
   BinOp(operator: String, left: Expr, right: Expr) extends Expr

def describe(e: Expr): String = (e: @unchecked) match {
   case Number(_) => "a number"
   case Var(_) => "a variable"
}
```

ako nema ove anotacije

```
warning: match is not exhaustive!
  missing combination UnOp
  missing combination BinOp
```

Tip Option

- `Option` tip omogućava "opcione" vrednosti
- Opciona vrednost ima dva oblika
  - `Some(x)`, gde je `x` stvarna vrednost
  - `None`, kada nema vrednost (nedostajuća vrednost)
- Jedna od mogućih primena: kolekcije
  - Map vraća `Some(vrednost)` za dati `ključ`, ako u mapi postoji ulaz `(ključ, vrednost)`
  - U suprotnom vraća `None`

Tip Option

- U kontekstu uparivanja obrazaca
  - rešava problem interpretacije vrednosti ako je nedostajuća
- Primer:

```
val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")
def show(x: Option[String]) = x match {
   case Some(s) => s
   case None => "?"
}

scala> show(capitals get "Japan")
res25: String = Tokyo

scala> show(capitals get "France")
res26: String = Paris

scala> show(capitals get "Mordor")
res27: String = ?
```

Tip Option

- Zamena za `null` u jezicima poput Jave
  - Primetiti: `Map[Int, Int]` ne može da uskladišti `null`
  - Dakle, ne može da vrati `null` ako ključ ne postoji (`null` nije saglasan vrednosnim tipovima)
  - Ali može da vrati `None`

- Prednost?
  - Za vrednost čiji je tip `String`, jednostavno je zaboraviti da može imati vrednost `null`
  - Čitanjem koda je jasnije da vrednost tipa `Option[String]` može biti `None`
  - Promenljiva tipa `Option[String]` ne može se koristiti kao `String`
    - na taj način, izostanak provere vrednosti postaje greška u prevođenju

Obrasci u for izrazima

- Uparivanje promenljivih u sastavu jedne torke

```
scala> val myTuple = (123, "abc")

scala> val (number, string) = myTuple
number: Int = 123
string: java.lang.String = abc
```

- Korisno ako se iterira kroz kolekciju torki

```
scala> for ((country, city) <- capitals)
   println("The capital of "+ country +" is "+ city)
The capital of France is Paris
The capital of Japan is Tokyo
```

- Šta ako se ne mogu upariti svi elementi kolekcije?

Obrasci u for izrazima

```
scala> val results = List(Some("apple"), None, Some("orange"))
results: List[Option[java.lang.String]] =
         List(Some(apple), None, Some(orange))

scala> for (Some(fruit) <- results)
   println(fruit)
apple
orange
```

- U ovom primeru, uparuju se samo oni elementi kolekcije koji se uparuju sa `Some(x)`
  - `None` se preskače