Funkcionalno programiranje

Kontrolne strukture

Uvod

- Poziv funkcije je jedna vrsta kontrolnih struktura
- Scala obezbeđuje druge „klasične“ kontrolne strukture:
  - for
  - while
  - if
  - try
- Posebna kontrolna struktura selekcije (pattern matching)
  - match
- **Ne postoje** naredbe za prekid ciklusa:
  - ✘ break
  - ✘ continue

Uvod

- Zbog funkcionalne orijentacije jezika, gotovo sve kontrolne strukture imaju rezultat
  - kod je jednostavniji
  - nisu potrebne posebne pomoćne promenljive za „hvatanje“ rezultata izračunatog u kontrolnoj strukturi
- Zahvaljujući funkcijama višeg reda, programer može da formira svoje kontrolne strukture
  - sintaksno izgledaju kao da su deo jezika
  - zapravo su funkcije višeg reda
  - ako su dovoljno generalne, mogu postati bibliotečke funkcije

if

- Ponašanje saglasno očekivanom (iz imperativnih jezika)
- Ima rezultat – ponaša se kao ternarni operator (C/C++)
- Primer (imperativni stil)

```
var filename = "default.txt"
if (!args.isEmpty)
        filename = args(0)
```

- Primer (funkcionalni stil)

```
val filename =
        if (!args.isEmpty) args(0)
        else "default.txt"
```

while

- Ciklus se ponaša na „klasičan“ način
- Nije izraz – nema smislen rezultat
  - preciznije, tip rezultata je Unit
- Čisti funkcionalni jezici nemaju cikluse
  - Scala zadržava ciklus zato što postoje situacije kada su praktičniji za upotrebu od konstrukta FP
  - Bez ciklusa bi morala da se koristi rekurzija, što može biti je nepraktično i slabije razumljivo
- Preporuka: pokušati pisanje koda bez while ciklusa
  - poput razlike između val i var
- Postoji i do – while ciklus

while

- Primer (imperativni stil)

```
def gcdLoop(x: Long, y: Long): Long = {
    var a = x
    var b = y
    while (a != 0) {
        val temp = a
        a = b % a
        b = temp
    }
    b
}
```

- Primer (funkcionalni stil)

```
def gcd(x: Long, y: Long): Long =
        if (y == 0) x else gcd(y, x % y)
```

for

(poglavlje 23)

- Ima više formi
  - iteracija kroz kolekcije
  - iteracija kroz opseg vrednosti (Range - posebna vrsta kolekcije)
  - iteracija kroz višestruke kolekcije, uz filtriranje

- Primer (iteracija kroz kolekcije)

```
val filesHere = (new java.io.File(".")).listFiles
      for (file <- filesHere)
            println(file)
```

  - Sintaksno, izraz "`file <- filesHere`" zove se ***generator***
  - Operator `<-` u ovom slučaju može da se čita kao „u“

for

- Primer (višestruki generator)
  - kasniji generator brže „varira“

```
for( x <- List(1, 2); y <- List("One", "Two") )
       print(x, y)
```

```
(1,One)(1,Two)(2,One)(2,Two)
```

for

- Opseg (Range)
  - `Range(x, y)` - pravi opseg `x, x+1, ..., y-1`
  - `x until y` - pravi opseg `x, x+1, ..., y-1`

  } immutable.Range

  - `x to y` - pravi opseg `x, x+1, ..., y`
  - `x to y by z` - pravi opseg `x, x+z, ..., x+Nz<y`

  immutable.Range.inclusive

- Primer

```
for (i <- 1 to 4)   // for(i <- 10 to 1 by -1)
      println("Iteration "+ i)

for (i <- 1 until 5)
      println("Iteration "+ i)

// Izbegavati
for (i <- 0 to filesHere.length-1)
      println(filesHere(i))
```

for

- Filtriranje
  - moguće je dodati `if` klauzulu unutar izraza u "for" zagradama
  - takva klauzula se zove *filtar*
- Primer – bez filtra (imperativni stil)

```
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere)
      if (file.getName.endsWith(".scala"))
            println(file)
```

- Kada se koristi filtar

```
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere if file.getName.endsWith(".scala"))
   println(file)
```

for

- Primer sa više `if` klauzula

```
for ( file <- filesHere
      if file.isFile
      if file.getName.endsWith(".scala")
    )     println(file)
```

- Analiza
  - prethodni primeri ne pokazuju veći kvalitet pristupa koji koristi filtar u odnosu na onaj koji ga ne koristi
  - ipak, bez filtra, rezultat `for` izraza je ()
  - `for` izraz može da formira i kao rezultat dâ novoformiranu kolekciju

for

- Ugneždene iteracije (višestruki generatori u `for` izrazu)

```
def fileLines(file: java.io.File) =
   scala.io.Source.fromFile(file).getLines.toList

def grep(pattern: String) =
      for (  file <- filesHere
             if file.getName.endsWith(".scala");
             line <- fileLines(file)
             if line.trim.matches(pattern)
          ) println(file +": "+ line.trim)

grep(".*gcd.*")
```

Zato što su generatori i filtri u običnim zagradama

Primetiti: dva puta se poziva `line.trim`

for

- Uspostavljanje (definisanje) novih imena

```
def grep(pattern: String) =
   for { file <- filesHere
         if file.getName.endsWith(".scala")
         line <- fileLines(file)
         trimmed = line.trim     // val
         if trimmed.matches(pattern)
       } println(file +": "+ trimmed)

grep(".*gcd.*")
```

for

- Formiranje nove kolekcije
  - prethodni primeri nisu vraćali koristan rezultat
  - umesto obrade elementa, moguće je izdvojiti ga u novu kolekciju
  - kolekcija će biti istog tipa kao i kolekcija kroz koju se iterira
  - koristi se rezervisana reč `yield`
  - sintaksa: `for` *klauzule* `yield` *telo*

```
def scalaFiles = for { file <- filesHere
                       if file.getName.endsWith(".scala")
                     } yield file

def test = for { i <- 1 to 5
                 if i % 2 == 0
                 j <- 10 to 11
               } yield i*j    // Vector(20, 22, 40, 44)
```

try / catch / finally

- Ponašanje izuzetaka isto kao u mnogim jezicima
  - funkcija/metoda može da završi bacanjem izuzetka, bez rezultata
  - pozivajuća funkcija/metoda može da
    - uhvati izuzetak
    - propagira izuzetak na viši nivo
- Bacanje: `throw`
  - throw **je izraz** koji ima rezultat tipa `Nothing`
  - razlog: jednostavnije formulisanje iskaza
- Primer:

```
val half =
   if (n % 2 == 0)
       n / 2
   else
       throw new RuntimeException("n mora biti paran")
```

try / catch / finally

- Hvatanje bačenih izuzetaka

```
try {
   telo try bloka
}
catch {
   case e: TipIzuzetka1 => TeloRukovaoca1
   case e: TipIzuzetka2 => TeloRukovaoca2
   case _: Throwable => TeloOpstegRukovaoca
}
```

- Scala ne zahteva `throws` klauzulu ili hvatanje proverenih izuzetaka
  - postoji `@throws` anotacija, ali nije obavezna

try / catch / finally

- Klauzula `finally`

```
import java.io.FileReader
val file = new FileReader("ulaz.txt")
try {
   // Procitati datoteku
}
finally {
   file.close() // Osloboditi ne-memorijski resurs
}
```

- Ponaša se isto kao u Javi
  - telo klauzule `finally` će uvek biti izvršeno, nezavisno od toga da li je izuzetak bačen

try / catch / finally

- Ima rezultat
  - ako se iz `try` bloka ne baci izuzetak: rezultat `try` klauzule
  - ako se izuzetak baci i uhvati: rezultat odgovarajuće `catch` klauzule
  - inače: bez rezultata
- Ignoriše se rezultat klauzule `finally`

```
def f(): Int = try return 1 finally return 2 // 2

def g(): Int = try 1 finally 2 // 1
```

match

- Mehanizam selekcije, sličan `switch` iz Jave
- Ima generalniji oblik od oblika prikazanog ovde (kasnije)

```
val firstArg = if (args.length > 0) args(0) else ""
firstArg match {
   case "salt" => println("pepper")
   case "chips" => println("salsa")
   case "eggs" => println("bacon")
   case _ => println("huh?")
}
```

- Selekcija se može vršiti na osnovu raznorodnih stavki (ne mora samo ceo broj, nabrajanje ili String)
- Ne zahteva `break` (nema automatskog "propadanja")
- Glavna razlika u odnosu na `switch`: **ima rezultat**

match

- Primer kada se koristi vrednost `match` izraza

```
val firstArg = if (args.length > 0) args(0) else ""
val friend =
   firstArg match {
       case "salt" => "pepper"
       case "chips" => "salsa"
       case "eggs" => "bacon"
       case "lemonade" | "coffee" => "sugar"
       case _ => "huh?"
   }

println(friend)
```

Ciklusi bez break / continue

- Smenom
  - `continue` se menja sa `if`
  - `break` se menja sa delom uslova (`while`) ciklusa

```
int i = 0;
boolean foundIt = false;
while (i < args.length) {
  if(args[i].startsWith("-")){
        i = i + 1;
        continue;
  }
  if(args[i].endsWith(".scala")){
        foundIt = true;
        break;
    }
    i = i + 1;
}
```

```
var i = 0
var foundIt = false
while(i<args.length && !foundIt){
  if (!args(i).startsWith("-")) {
    if(args(i).endsWith(".scala"))
      foundIt = true
  }
  i = i + 1
}
```

Ciklusi bez break / continue

- Neki slučajevi se mogu rešiti rekurzijom

```
var i = 0
var foundIt = false
while(i<args.length && !foundIt){
  if (!args(i).startsWith("-")) {
    if(args(i).endsWith(".scala"))
      foundIt = true
  }
  i = i + 1
}
```

```
def searchFrom(i: Int): Int =
  if (i >= args.length) -1
  else if(args(i).startsWith("-"))
     searchFrom(i + 1)
  else if
    (args(i).endsWith(".scala")) i
  else searchFrom(i + 1)

val i = searchFrom(0)
```

- Ciklus je zamenjen rekurzivnim pozivom (terminalna rek.)
- Rezultujuća rekurzivna funkcija ima smisleno ime
- Postoji „break“ – realizovan bacanjem izuzetka (pogl. 7.6)

Pravljenje programski definisanih kontrolnih struktura

- Nisu „prave“ kontrolne strukture
- Sintaksno deluju kao da su u pitanju k.s. ugrađene u jezik
- Koriste se funkcije višeg reda:
  funkcije kojima se prosleđuju funkcije
- Primer: funkcija koja bezbedno upravlja resursom
  i obavlja proizvoljan posao

```
def withPrintWriter(file: File, op: PrintWriter => Unit) {
    val writer = new PrintWriter(file)
    try{
        op(writer)
    } finally {
        writer.close()
    }
}
```

```
withPrintWriter(
   new File("date.txt"),
   writer =>
    writer.println(new java.util.Date)
)
```

Pravljenje programski definisanih kontrolnih struktura

- Cilj je (sintaksno) izbeći pozivanje funkcije sa 2 parametra
- Cilj je uvesti vitičaste zagrade u poziv funkcije
  - vitičaste zagrade su moguće samo ako postoji jedan parametar

```
def withPrintWriter(file: File)(op: PrintWriter => Unit) {
    val writer = new PrintWriter(file)
    try {
        op(writer)
    } finally {
        writer.close()
    }
}
```

```
val file = new File("date.txt")
withPrintWriter(file) {
    writer =>
     writer.println(new java.util.Date)
}
```

Kontrolne strukture i prenos po imenu

- Neke kontrolne strukture (`if`, `while`) ne zahtevaju prenos dodatnih argumenata (poput `writer` iz primera)
- Assert je takva struktura
  - ako je provera aktivirana – proverava validnost iskaza
  - ako nije – ignoriše

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
     throw new AssertionError
```

```
myAssert(() => 5 > 3)
```

Ne može samo

```
myAssert(5 > 3)
```

zato što se očekuje funkcija.
A to je upravo oblik koji je poželjan!

Kontrolne strukture i prenos po imenu

- Prenos po imenu rešava problem!

```
def byNameAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError


byNameAssert(5 > 3)
```

Zašto prvobitno ovo rešenje (prenos po vrednosti) nije dobro?

```
def boolAssert(predicate: Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

Vrednost parametra se računa PRE poziva!

Potencijalni bočni efekti čak i kada Assert treba ignorisati.