Cheat Sheet Laboratoare Scala

Cuprins

Recursitivitatea pe coada `@tailrec`

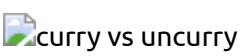
- doar autopel de functie
- nu au loc alte operatii cu autoapelul de functii (doar in parametrii lui)
- se foloseste un acumulator

Functii de ordin superior (`High Order Functions`)

- pot primi alte functii ca parametru
- pot intoarce alte functii
- inchidere functionala

Linkuri utile:

- YouTube Curried Functions - Computerhile
- YouTube High Order Functions


curry vs uncurry

Tipuri de functii:

- `uncurry` : primeste toti parametri odata

```
def f(x: Int, y: Int, z: Int) = x + y + z
```

- `curry` : primeste parametrii pe rand

```
def f(x: Int)(y: Int)(z: Int): Int = x + y + z
```

 curry high order function

List

Listele sunt `immutabile`, insemnand ca nu pot fi modificate odata ce au fost create.

Lista vida: `List()`

Operatorii pentru liste:

- `Nil`
  - Descriere: lista vida
- `::`
  - **Descriere**: este operatorul de constructie al listei si este folosit pentru a adauga un element la `inceputul` unei liste
  - **Sintaxa**: `el :: list`
- `+:`
  - **Descriere**: primeste un element si o lista, intoarce o noua lista care contine elementul la `inceputul` listei
  - **Sintaxa**: `el +: list`
- `:+`
  - **Descriere**: primeste o lista si un element intoarce o lista noua, inserand elementul la `sfarsitul` listei
  - **Sintaxa**: `list :+ el`
- `:::`
  - **Descriere**: primeste doua liste, pe care le conateneaza
  - **Sintaxa**: `list1 ::: list2`
- `++`
  - **Descriere**: echivalent cu operatorul `:::` primeste doua liste si returneaza list concatenata

    - **Sintaxa**: `list1 ++ list2`

- `list(idx)` =
    - **Descriere**: accesarea unui element din lista se poate face prin indice, scris intre paranteze rotunde

Operator	Sintaxa
`::`	`el :: list`
`:+`	`list :+ el`
`+:`	`el +: list`
`:::`	`list1 ::: list2`
`++`	`list1 ++ list2`

```
val list01: List[Int] = Nil                 // Empty List
val list02: List[Int] = List.empty          // Empty List
val list03 = List.empty[Int]                // Empty List
val list04 = List(1, 2, 3, 4)               // List Constructor
val list05 = 1 :: 2 :: 3 :: 4 :: Nil        // Cons
val list06 = 1 :: List(2, 3, 4)             // Cons and List Constructor
val list07 = 101 :: List(1, 2, 3)           // [101, 1, 2, 3]
val list08 = 101 +: List(1, 2, 3)           // [101, 1, 2, 3]
val list09 = List(1, 2, 3) :+ 101           // [1, 2, 3, 101]
val list10 = List(1, 2) ::: List(3, 4)      // [1, 2, 3, 4]
val list11 = List(1, 2) ++ List(3, 4)       // [1, 2, 3, 4]

val first = list11(0)                        // indexing
```

Metode pentru liste:

Nume functie	Descriere
`foldLeft`	Reduce lista la o singura valoare, aplicand o functie de la stanga la dreapta.
`foldRight`	Reduce lista la o singura valoare, aplicand o functie de la dreapta la stanga.
`fold`	Reduce lista la o singura valoare, pornind de la o valoare initiala.
`map`	Aplica o functie pe fiecare element al listei si intoarce o noua lista cu rezultatele functiei.
`filter`	Primeste o functie care intoarce `Boolean` si intoarce elementele listei pentru care evaluarea functiei intoarce `true`.
`filterNot`	Primeste o functie care intoarce `Boolean` si intoarce elementele listei pentru care evaluarea functiei intoarce `false`.

Nume functie	Descriere
`zip`	Imperecheaza doua liste. Primeste o lista si intoarce o **lista de tupluri**: `._1` din prima lista, `._2` din a doua lista.
`flatMap`	Aplica o functie care intoarce o colectie pe fiecare element si aplatizeaza rezultatele intr-o singura lista.
`reduceLeft`	Similar cu `foldLeft`, dar fara valoare initiala; reduce lista la o singura valoare, aplicand functia de la stanga la dreapta.
`reduceRight`	Similar cu `foldRight`, dar fara valoare initiala; reduce lista la o singura valoare, aplicand functia de la dreapta la stanga.
`partition`	Imparte lista in doua liste pe baza unei functii care intoarce `Boolean`: una cu elementele pentru care functia intoarce `true`, alta pentru care intoarce `false`.
`find`	Gaseste primul element care indeplineste o conditie specificata de o functie.
`forall`	Verifica daca toate elementele listei indeplinesc o conditie specificata de o functie.
`exists`	Verifica daca cel putin un element din lista indeplineste o conditie specificata de o functie.
`take`	Intoarce primele `n` elemente din lista.
`drop`	Elimina primele `n` elemente din lista si intoarce restul listei.
`head`	Intoarce primul element al listei.
`tail`	Intoarce lista fara primul element.
`init`	Intoarce lista fara ultimul element.
`last`	Intoarce ultimul element al listei.

Pattern `match`-ing pe liste

Operatorul `::` este cunoscut si ca operatorul `Cons` (`Cons(x, xs)`)

- `x` = primul element al listei
- `xs` = restul listei

```
def sum(l: List[Int]): Int = {
  // pattern matching
  l match {
    case Nil => 0
    case x :: xs => x + sum(xs)
  }
}
```

`FoldLeft`

- este o functie care itereaza o lista, pe care o reduce la un singur element
- `( (acc + 1) + 2 ) + 3`
- prima operatie are loc intre: primul element si acumulator
- parcurge lista **de la stanga la dreapta**
- este `@tailrec`

Functia default din biblioteca Scala `foldLeft` este o **functie de ordin superior (high order function)** si are urmatoarea semnatura:

```
/**
 * @param z     valoarea initiala a acumulatorului
 * @param op    functie intre `acumulator si valoare`
 *              primeste o `pereche` dintre `acumulator si valoare`
 *                - primul arg: `acc`
 *                - al doilea arg: `value`
 *              intoarce noul acumulator
 * @tparam A    tipul de date al lui `value` (elementele din lista)
 * @tparam B    tipul de date al elemeentelor listei
 * @return      valoarea la care a fost redusa lista
 */
def foldLeft[B](z: B)(op: (B, A) => B): B
```

Exemplu:

```
val result: Int = List(1, 2, 3).foldLeft(init_acc)((acc, value) => acc +
value)
```

```
val list = List(1, 2, 3, 4)

// foldLeft: (0 + 1) + 2 + 3 + 4
val sumLeft = list.foldLeft(0)(_ + _)
println(sumLeft) // Output: 10
```

FoldRight

- este o functie care itereaza o lista, pe care o reduce la un singur element
- `1 + (2 + (3 + acc) )`
- prima operate are loc intre: ultimul element si acumulator
- parcruge lista **de la dreapta la stanga**

Functia defaut din biblioteca Scala `foldRight` este o **functie de ordin superior** si are urmatoarea semnatura:

```
/**
 * @param z     valoarea initiala a acumulatorului
 * @param op    functie intre `valoare si acumulator`
 *              primeste o pereche cu `valoarea si acumulatorul`
 *                - primul arg: `value`
 *                - al doilea arg: `acc`
 *              returneaza noul acumulator
 * @tparam A    tipul de date al lui `value` (elementele din lista)
 * @tparam B    tipul de date al elementelor listei
 * @return      valoarea la care a fost redusa lista
 */
def foldRight[B](z: B)(op: (A, B) => B): B
```

Exemplu:

```
val result: Int = List(1, 2, 3).foldRight(init_acc)((value, acc) => acc + value)
```

```
val list = List(1, 2, 3, 4)

// foldRight: 1 + (2 + (3 + (4 + 0)))
val sumRight = list.foldRight(0)(_ + _)
println(sumRight) // Output: 10
```

`FoldLeft` vs `FoldRight`

	`foldLeft`	**`foldRight`**
Recursivitate	pe coada (`@tailrec`)	pe stiva
Antetul functiei	`def foldLeft[B](z: B)(op: (B, A) => B): B`	`def foldRight[B](z: B)(op: (A, B) => B): B`
Param1	`z`	`z`
Tip param 1	`B`	`B`
Descr. param 1	**valoarea initiala a acumulatorului**	**valoarea initiala a acumulatorului**
Param 2	`op`	`op`
Tip param 2	`(B, A) => B`	`(A, B) => B`
Descr. param 2	functie de reducere	functie de reducere
Ordinea parametrilor `functiei de reducere`	**`acc val`**	**`val acc`**

```scala
val sum1: Int = List(1, 2, 3).foldLeft(0)((acc, value) => acc + value)
val sum2: Int = List(1, 2, 3).foldRight(0)((value, acc) => acc + value)

println(sum1)     // 6
println(sum2)     // 6

val reversed1: List[Int] = List(1, 2, 3).foldLeft(List.empty)((acc, value)
=> value +: acc)
val reversed2: List[Int] = List(1, 2, 3).foldRight(List.empty)((value, acc)
=> acc :+ value)

println(reversed1)    // List(3, 2, 1)
println(reversed2)    // List(3, 2, 1)


// pentru claritate, introducem tipuri de date in expresiile `lambda`
val sum3: Int = List(1, 2, 3).foldLeft(0)((acc: Int, value: Int) => acc +
value)
val sum4: Int = List(1, 2, 3).foldRight(0)((value: Int, acc: Int) => acc +
value)

println(sum3)     // 6
println(sum4)     // 6

val reversed3: List[Int] = List(1, 2, 3)
  .foldLeft(List.empty)((acc: List[Int], value: Int) => value +: acc)
val reversed4: List[Int] = List(1, 2, 3)
  .foldRight(List.empty)((value: Int, acc: List[Int]) => acc :+ value)

println(reversed3)    // List(3, 2, 1)
println(reversed4)    // List(3, 2, 1)
```

Fold

```scala
/**
 *
 * @param z     valoarea initiala a acumulatorului
 * @param op    este o functie intre `acumulator si valoare`
 *              primeste o pereche intre `acumulator si valoare`
 *                - primul arg: `acc`
 *                - al doilea arg: `value`
 *              si returneaza noul acumulator
 * @tparam A    tipul de date al lui `value` (elementele din lista)
 * @tparam B
 * @return
 */
def fold[B](z: B)(op: (B, A) => B): B
```

Exemplu:

```
val result: Int = List(1, 2, 3).fold(init_acc)((acc, value) => acc + value)
```

`mkString` (conversia unei liste la un `String`)

> vine de la `make String`

Functia `mkString` converteste elementele unei colectii, spre exemplu o lista, intr-un singur sir (`String`).

Aceasta permite specificarea unui separator intre elemente si, optionale, prefix si sufix pentru sirul rezulat.

Functia `mkString` din Scala are urmatoerele antete (semnaturi):

```
// fara argumente
def mkString: String


/**
 * @param sep   seperator
 */
def mkString(sep: String): String


/**
 * @param start   prefix
 * @param sep     separator
 * @param end     sufix
 */
def mkString(start: String, sep: String, end: String): String
```

Exemple:

```
val list = List(1, 2, 3)
val result1 = list.mkString                    // "123"
val result = list.mkString(", ")               // "1, 2, 3"
val result = list.mkString("(", ", ", ")")     // "(1, 2, 3)"
```

> `mkString` va apela metoda `toString` daca intalneste obiecte

Alte metode pentru liste

```
val nums = List(1, 2, 3, 4, 5)

// `head` = primul element din lista ; echivalentul lui `x` din `list ==
```

```
Cons(x, xs) == x :: xs`
val first: Int = nums.head                    // 1

// `tail` = lista fara primul element ; echivalentul lui `xs` din `list ==
Cons(x, xs) == x :: xs`
val withoutFirst: List[Int] = nums.tail       // [2, 3, 4, 5]


val sum: Int = nums.reduce(_ + _)             // 15
val evens = nums.filter(_ % 2 == 0)           // [2, 4]
val odds = nums.filterNot(el => el % 2 == 0)  // [1, 3, 5]

// `map` aplica o functie fiecarui element din lista
val signs = nums.map(el => el % 2)            // [1, 0, 1, 0, 1]


val letters = List("a", "b", "c", "d")

// `zip` combina doua liste, formand o lista de perechi (tuple)
// pairs1 == List((1,a), (2,b), (3,c), (4,d))
val pairs1: List[(Int, String)] = nums.zip(letters)

// rezultatul functiei `zip` va avea dimensiunea celei mai scurte liste
// pairs2 == pairs3 == List((1,a), (2,b))
val pairs2 = List(1, 2, 3, 4, 5).zip(List("a", "b"))
val pairs3 = List(1, 2).zip(List("a", "b", "c", "d", "e"))


// [1, 0, 1, 0, 1]
val list1 = nums.zip(signs).map { case (a, b) => a * b }

// list2 == [0, 2, 2, 4, 4]
val list2 = nums.zip(signs).map( {case (a, b) => a - b} )

// list3 == [0, -2, -2, -4, -4]
val list3 = signs.zip(nums).map( {case (a, b) => a - b} )
```

Tipul de date `Option`

> Este folosit pentru propagarea erorilor

Tipul de date `Option` este un *container / cutie* care include un singur element (`Some(_)`) sau niciunul (`None`).

Folosim `Option` pentru a scrie functii robuste, in cazul in care returneaza `null` sau nu reusesc sa returneze o valoare acceptata.

```
def divide(x: Double, y: Double): Option[Double] = {
  if (y == 0.0) None
  else Some(x / y)
```

```
}

val div1: Option[Double] = divide(0, 0)         // None
val div2: Option[Double] = divide(5.0, 2.5)     // Some(2.0)

val div3 = divide(0, 0).get                          //
NoSuchElementException
val div4: Double = divide(5.0, 2.0).get              // 2.5

// in caz de `None`, se va returna valoarea din `getOrElse`
val div5: Double = divide(0, 0).getOrElse(-1.4)       // -1.4
val div6: Double = divide(5.0, 2.0).getOrElse(-1.99)  // 2.5


// pattern-matching an `Option`
divide(5.0, 0.0) match {
  case None => println("Division by zero!")
  case Some(x) => println("Res = " + x)
}

// pattern-matching an `Option`
divide(5.0, 2.0) match {
  case None => println("Division by zero!")
  case Some(x) => println("Res = " + x)
}
```

Tuplu

In Scala, un tuplu (tuple) este o structura de date care contine un numar fix de elemente, acestea putand avea tipuri diferite.

> Scala ofera suport pentru tupluri de la 1 pana la 22 de elemente

Tuplurile sunt imutabile, ceea ce inseamna ca, odata ce au fost create, valorile nu pot fi schimbate.

Pentru a defini un tuplu, folosim paranteze rotunde si separam elementele prin virgula.

Pentru a accesa un element din tiplu, folosim numele tuple-ului, simbolurile `._` urmate de indexul elementului (un numar natural intreg).

> ex: `val first = t._1`
>
> ex: `val second = t._2`

```
val tuple2: (Int, String)  = (42, "Scala")
val tuple3: (Double, Int, String) = (2.5, 42, "Hello")
val tuple4 = (1, 2, 3, 4)

println(tuple2._1)
println(tuple2._2)
```

```scala
val v1: Double = tuple3._1
val v2: Int = tuple3._2
val v3: String = tuple3._3

tuple4 match {
  /** folosim pattern matching pentru a destructura elementele unui tuplu
**/
  case (nr1, nr2, nr3, nr4) => println(nr1 + nr2 + nr3 + nr4)   // 1 + 2 +
3 + 4 == 10
}



type GraphAdjLists = List[(Int, List[Int])]

val graph: GraphAdjLists = List(
  (1, List(3, 5, 6)),
  (2, List(4, 6)),
  (3, List(2)),
  (4, List(1, 5, 6))
)

val str: String = graph
      /** destructurare element **/
  .map( { case (node, neighs) => s"$node -> " ++ neighs.mkString(" ") } )
  .mkString("\n")   // `mkString` converteste o lista la un string

println(str)
// 1 -> 3 5 6
// 2 -> 4 6
// 3 -> 2
// 4 -> 1 5 6
```

Utilizarea `case`-ului in interiorul functiei `map`

In Scala, atunci cand lucram cu colectii de tip `perechi` (**tupluri**), putem folosi `pattern matching` pentru a **deconstrui** fiecare element al colectiei in componentele sale.

```scala
val list: List[(Int, String)] = List((1, "unu"), (2, "doi"), (3, "trei"))

val newList = list.map { case (nr, str) => (nr * 2, str.toUpperCase) }
// List((2, "UNU"), (4, "DOI"), (6, "TREI"))
```

In exemplul de mai sus:

- `list.map` parcurge o lista si aplica o functie asupra fiecarui element al ei
- `{ case (nr, str) => ... }` blocde pattern matching pt a deconstrui fiecare tuplu

Alte exemple:

```scala
val pairs = List(("a", "apple"), ("b", "banana"))

val uppercasedPairs = pairs.map { case (letter, fruit) =>
  (letter.toUpperCase, fruit.capitalize)
}
```

```scala
val numberPairs: List[(Int, Int)] = List((1, 2), (3, 4))

val summedPairs: List[Int] = numberPairs.map { case (a, b) =>
  a + b
}
```

Functie partiala

In cazul in care lista contine diverse tipuri de date, trebuie sa folosim niste `case`-uri speciale, nimite `functii partiale`.

Ele sunt de obicei folosite impreuna cu pattern matching pentru a deconstrui si manipula datele in functie de forma lor specficia.

Exemple:

```scala
val variousPairs: List[(Any, Any)] = List((1, 2), ("a", "b"), (3.0, 4.0),
(true, false))

val handledPairs: List[(Any, Any)] = variousPairs.map {
  case (a: Int, b: Int) => (a + b, a * b)
  case (a: String, b: String) => (a + b, a.reverse + b.reverse)
  case (a: Double, b: Double) => (a / b, a * b)
  case (a: Boolean, b: Boolean) => (a || b, a && b)
  case other => other
}

// handledPairs va fi List((3, 2), ("ab", "aabb"), (0.75, 12.0), (true,
false))
```

```scala
val optionPairs: List[(Option[Int], Option[String])] = List((Some(1),
Some("unu")), (None, Some("doi")), (Some(3), None))

val handledOptionPairs: List[(Option[Int], Option[String])] =
optionPairs.map {
  case (Some(num), Some(text)) => (Some(num * 2), Some(text.toUpperCase))
  case (None, Some(text)) => (None, Some(text.reverse))
  case (Some(num), None) => (Some(num + 10), None)
  case other => other
```

```
}

// handledOptionPairs va fi List((Some(2), Some("UNU")), (None,
Some("iod")), (Some(13), None))
```

Expresii `lambda` (functii anonime)

> Sintaxa

```
(param1: Type1, param2: Type2) => expression
```

In Scala, tipul valorii de retur al unei functii lambda este dedus automat de catre compilator pe baza tipurilor parametrilor si a corpului functiei.

De aceea, nu este nevoie sa specificam explici tipul de retur

Componentele unei epxresii `lambda`:

- **Parametrii**: Lista de parametri in paranteze, fiecare parametru avand un nume si un tip (ex.: `param1: Type1`).

  > 💡 Daca exista doar un singur parametru, parantezele pot fi omise

  > **!!!** Atunci cand specificam tipul de date al unui **parametru tuplu** intr-o expresie `lambda`, parametrii trebuie sa fie trecuti intre paranteze.

  > 💡 Parametrii tuplu al unei expresii `lambda` pot fi **destructurati** folosind `case`

  > **!!!** Atunci cand specificam tipul de date al unui parametru ca fiind functie, toti parametrii (indiferent daca este doar unul singur) trebuie sa fie trecuti intre paranteze rotunde.

- **Sageata `=>`**: separa parametrii de corpul functiei
- **Corpul expresiei**

```
// expresie lambda cu un parametru
x: Int => x * 2
x => x * 2
_ => _ * 2
```

```
// expresie lambda cu doi parametri
(x: Int, y: Int) => x + y
(x, y) => x * y
```

```
def doubler(): Int => Int = {
  (n: Int) => 2 * n   // expresie lambda (functie anonima)
}
```

```
// asociem o exprie `lambda` (functie anonima) unei variabile
val lambda_cond = (name: String, grade: Int) => grade >= 5
```

Expresii `lambda` cu `tupluri`

> **!!!** Atunci cand specificam tipul de date al unui **parametru tuplu** intr-o expresie `lambda`, parametrii trebuie sa fie trecuti intre paranteze.

> 💡 Parametrii tuplu al unei expresii `lambda` pot fi **destructurati** folosind `case`

```
// expreise lambda care primeste un tuplu (pereche de elemente)
t:
t => t._1 + t._2
_ => _._1 + _._2
```

```
val addTuple1 = (t: (Int, Int)) => t._1 + t._2
val addTuple2 = { case (x: Int, y: Int) => x + y }

/// !!! Acest cod va produce o eroare de sintaxă
/// !!! val addTuple = t: (Int, Int) => t._1 + t._2

val tuplesList = List((1, 2), (3, 4), (5, 6))
val sums1 = tuplesList.map((t: (Int, Int)) => t._1 + t._2)
val sums2 = tuplesList.map { case (x: Int, y: Int) => x + y }
```

Expresii `lambda` cu `functii de ordin superior`

> **!!!** Atunci cand specificam tipul de date al unui parametru ca fiind functie, toti parametrii (indiferent daca este doar unul singur) trebuie sa fie trecuti intre paranteze rotunde.

> 🖇 **Reminder:** O functie de ordin superior (high order functions) este o functie pentru care primeste / returneaza o functie

```
// `f` este o functie
val applyFunction = (f: Int => Int, x: Int) => f(x)
val applyFunction = (f, x) => f(x)
```

Tipuri de date in Scala

```scala
// Numere Naturale
trait Nat
case object Zero extends Nat
case class Succ(x: Nat) extends Nat



// Arbori Binari
trait BTree
case object EmptyTree extends BTree
case class Node(value: Int, left: BTree, right: BTree) extends BTree



// Evaluarea expresiilor matematice de baza
trait Expr
case class Atom(a: Int) extends Expr
case class Add(e1: Expr, e2: Expr) extends Expr
case class Mult(e1: Expr, e2: Expr) extends Expr



// definierea unei matrici (lista de liste)
type Matrix = List[List[Int]]
```

```scala
// Arbori Binari
trait BTree
case object EmptyTree extends BTree
case class Node(value: Int, left: BTree, right: BTree) extends BTree



// preordine: Radacina - Stanga - Dreapta
def flatten_preordine(tree: BTree): List[Int] = {
  tree match {
    case EmptyTree => Nil // List.empty[Int]
    case Node(value, left, right) => {
      /** opeatorii `:::` si `++` concateneaza doua liste **/
      List(value) ::: flatten_preordine(left) ::: flatten_preordine(right)
    }
  }
}
```

```scala


// Evaluarea expresiilor matematice de baza
```

```scala
trait Expr
case class Atom(a: Int) extends Expr
case class Add(e1: Expr, e2: Expr) extends Expr
case class Mult(e1: Expr, e2: Expr) extends Expr


def evaluate(e: Expr): Int = e match {
  case Atom(a) => a
  case Add(e1, e2) => evaluate(e1) + evaluate(e2)
  case Mult(e1, e2) => evaluate(e1) * evaluate(e2)
}
```