Paradigma Calculul Functional

Cursul nr. 12

Mihai Zaharia

Cum a început.... (au fost o dată ca niciodată ...)

ipoteza Church

și conducătorii lor

de doctorat ...

ipoteza Turing

Java vs Kotlin

```
public class JavraLambda //ex 1
{   interface Test
        { void test(); }
    private static void apel_test(Test test)
        { test.test(); }
    public static void main(String[] args)
        { apel_test(()->{ System.out.println("apel metoda
apel_test()"); }); }
}
fun apel_test(test:()->Unit) //ex 2
{ test()  }
fun main(args: Array<String>)
{ apel_test({ println("apel functie apel_test()");})  }
```

Utilizarea unei funcții ca o proprietate

```
fun main(args: Array<String>)
{
    val dif_numere = { x: Int, y: Int -> x - y }
    println("Diferenta 1 este ${dif_numere(33,66)}")
    println("Diderenta 2 este ${dif_numere(77,11)}")
}
```

Sintaxa specifică funcțiilor Lambda în Kotlin

```
fun main(args: Array<String>) {
    val invers:(Int)->Int
    invers = {numar ->
        var n = numar
        var numarInvers = 0
        while (n>0) {
            val digit = n%10
            numarInvers=numarInvers*10+digit
            n/=10
        }
        numarInvers
    }
    println("inversul lui 123 ${invers(123)}")
    println("inversul lui 456 ${invers(456)}")
    println("inversul lui 789 ${invers(789)}")
}
```

si exemplul de executie

```
inversul lui 123 321
inversul lui 456 654
inversul lui 789 987

Process finished with exit code 0
```

Funcții de nivel superior

```
fun procesareNrPare(numar:Int,procesare:(Int)->Int):Int {
    if(numar%2==0) {
        return procesare(numar)
    } else {
        return numar
    }
}
fun main(args: Array<String>) {
    var nr1=4
    var nr2 = 5
    println("Apel cu ${nr1} si operatia (it*2): ${procesareNrPare(nr1,{it*2})}")
    println("Apel cu ${nr2} si operatia (it*2): ${procesareNrPare(nr2, {it*2})}")
}
```

Funcții de nivel superior

```
fun apelCuIntoarcere(n:Int):(String)->Char {
    return { it[n] }
}
fun main(args: Array<String>) {
    var pos = 4
    print("${apelCuIntoarcere(1)("abc")}\n")
    print("${apelCuIntoarcere(0)("def")}\n")
    try {
        print(apelCuIntoarcere(pos)("ghi"))
    }
    catch (e: StringIndexOutOfBoundsException) {
        print("Indexul ${pos} este in afara sirului")
    }
}
```

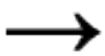
Efecte laterale

```
class operatiiAritmetice {
   var nr1:Int=0
   var nr2:Int=0
   fun suma(nr1:Int = this.nr1,nr2:Int =
this.nr2):Int {
      this.nr1 = nr1
      this.nr2 = nr2
      return nr1+nr2
   }
}
fun main(args: Array<String>) {
   var nr1 = 10
   var nr2 = 15
   val obCalcul = operatiiAritmetice()
   println("Suma dintre ${nr1} si ${nr2}
este ${obCalcul.suma(nr1,nr2)}")
}
```

```
class functionalOperatiiAritmetice {
   val sumaf: (Int, Int) -> Int = { x, y -> x + y }
   fun executaOperatia(x: Int, y: Int, op: (Int,
Int) -> Int): Int = op(x, y)
}
fun main(args: Array<String>) {
   var nr1 = 10
   var nr2 = 15
   val obFCalcul =
functionalOperatiiAritmetice()
   println("Suma dintre ${nr1} si ${nr2} este
${obFCalcul.executaOperatia(nr1,nr2,obC
alcul.sumaf)}")
 }
```

Efecte laterale

```
fun main(args: Array<String>) {
    var nr1 = 10
    var nr2 = 15
    val sumaf: (Int, Int) -> Int = { x, y -> x + y }//stil functional
    fun sumak(x: Int, y: Int) = x + y //stil kotlin
    fun executaOperatia(x: Int, y: Int, op: (Int, Int) -> Int): Int = op(x, y)
    println("Suma dintre ${nr1} si ${nr2} este ${executaOperatia(nr1,nr2,sumaf)}")
    println("Suma dintre ${nr1} si ${nr2} este ${executaOperatia(nr1,nr2,::sumak)}")
}
```



Funcții pure

```
fun suma(a:Int = 0,b:Int = 0):Int {
    return a+b
}
```

vararg

```
fun calculMedie(listaNumere: List<Int>): Float {
    var suma = 0.0f
    for (element in listaNumere) {
        suma += element
    }
    return (suma / listaNumere.size)
}
fun calculMedieParametri(vararg lista_parametri: Int):
Float {
    var suma = 0.0f
    for (element in lista_parametri) {
        suma += element
    }
    return (suma / lista_parametri.size)
}
fun <T> enumerareCaOLista(vararg parametri_intrare:
T): List<T> {
    val rezultat = ArrayList<T>()
    for (element in parametri_intrare)
        rezultat.add(element)
    return rezultat
}
```

```
fun main(args: Array<String>) {
    val tablou = arrayListOf(1, 2, 3, 4)
    val rezultat = calculMedie(tablou)
    print("\nMedia este ${rezultat}")
    val rezultat1 = calculMedieParametri(1, 2, 3)
    print("\nMedia1 este ${rezultat1}")
    val rezultat3 = enumerareCaOLista(1, 2, 3, 4, 5, 6, 7,
8, 9)
    print("\nTransformare enumerare in lista
${rezultat3}")
    print("\nMedia1 este
${calculMedieParametri(*rezultat3.toIntArray())}")
    val tablou1 = intArrayOf(1, 2, 3, 4)
    val rezultat4 = calculMedieParametri(5, 6, 7,8,9,
*tablou1)
    print("\nMedia1 este ${rezultat4}")
}
```

Parametri alias - Named parameters

```
typealias Kg = Double
typealias cm = Int
data class ClientBanca (val numeFamilie: String,
                        val numeMijlociu: String,
                        val numeMic: String,
                        val seriePasaport: String,
                        val greutate: Kg,
                        val inaltime: cm,
                        val semneParticulare: String)
fun main(args: Array<String>) {
   val client1 = ClientBanca("Mike", "Mouse", "Rabbit", "XX234837447", 82.3, 180, "nu are")
   val client2 = ClientBanca(
      numeMic = "Rabbit",
      numeMijlociu = "Mouse",
      numeFamilie = "Mike",
      seriePasaport = "xe4244rf33333",
      greutate = 100.0,
      inaltime = 180,
      semneParticulare = "nu are"
   )
print("\n${client1}")
print("\n${client2}")
}
```

alias cu vararg sau funcții de nivel superior

```
fun paramDupaVararg(nrCurs: Int, vararg lista_studenti: String, tempCamera: Double) {//ex1
    //corpul functiei
}
paramDupaVararg(688, "Gica", "Bula", "Andreea", "Veorica", tempCamera = 15.0)//apel
fun test(f: (Int, String) -> Unit) {//ex2
   f(1, "Bula")
}//cu apelul
test { q, w ->
    //procesare
}//poate fi rescrisa ca
fun test(f: (nume:String, varsta:Int) -> Unit) {
   f(“Strula", 10)
}//daca incerc insa ca mai jos
fun test(f: (nume:String, varsta:Int) -> Unit) {
   f(nume = "kati", virsta = 3, ) //eroare de compilare}
```

Funcții de extensie

```
fun String.trimitLaConsola() = println(this) // la tip
class Om(val nume: String)
fun Om.spune(): String = "${this.nume} spune Vai" //la clasa
fun main(args: Array<String>) {
    "lA examen".trimitLaConsola()
    val x=Om("Bula")
    x.spune().trimitLaConsola()
}
```

si rezultatul executiei

```
lA examen
Bula spune Vai

Process finished with exit code 0
```

Funcții de extensie și funcții membre

```
open class Canina {
   open fun vorbeste() = "Un animal din clasa Caninelor face: ham ham!"   }
fun mesajScris(canina: Canina) {
   println(canina.vorbeste())
}
class Caine : Canina() {
   override fun vorbeste() = "Un caine face: vauf vauf!" //modificare comportament in clasa derivata
}
fun mesajScris1(canina: Canina) {
   println(canina.vorbeste1())
}
fun Caine.vorbeste()="Din functia extensie Un caine face haf haf" //fiind functie de extensie nu supraincarca!!!
deci este ignorata!!!
fun Canina.vorbeste1() = "functie extensie specifica cainelui"
fun main(args: Array<String>) {
   mesajScris(Canina())
   mesajScris(Caine()) //baza polimorfismului
   mesajScris1(Caine()) //desi este definita la nivelul clasei canina prin mosternire poate fi apelata si in caine
}
```

Funcții de extensie și funcții membre

```
open class Primata(val name: String)
fun Primata.vorbeste() = "$name: uhaha uhaha"
open class MaimutaMare(name: String) : Primata(name)
fun MaimutaMare.vorbeste() = "${this.name} : urlet"
fun mesajScris(primata: Primata) {
   println(primata.vorbeste())
}
fun mesajScris1(maimutza: MaimutaMare) {
   println(maimutza.vorbeste())
}
fun main(args: Array<String>) {
   mesajScris(Primata("alex")) // apeleaza vorbeste din primata
   mesajScris(MaimutaMare("crrr")) // apeleaza vorbeste din primata
   mesajScris1(MaimutaMare("ciii")) // apeleaza vorbeste din maimuta
   mesajScris1(Primata("jiii") as MaimutaMare) // apeleaza vorbeste din maimutaeroare nu pot
converti primata la maimuta
}
```

Dispatch recv

```
open class Felina
open class Pisica():Felina()
open class Primata(val name: String)
fun Primata.vorbeste() = "$name: face uhaha uhaha"//extensie primate
open class MaimutaMare(name: String) : Primata(name)
fun MaimutaMare.vorbeste() = "${this.name} : urlet" //ignorata deoarece amm deja un vorbeste
din primata
fun mesajScris(primata: Primata) {    println(primata.vorbeste()) }
open class Ingrijitor(val name: String) {
   open fun Felina.react() = "HRRMR!!!"
   fun Primata.react() = "$name se joaca cu ${this@Ingrijitor.name}" //ditribuitor intern bagat aici
pentru ca altfel nu am acces la .name
   fun areGrija(felina: Felina) {        println("Felina face: ${felina.react()}")     }
   fun areGrija(primata: Primata){    println("Primata spune: ${primata.react()}")      }
   fun Ingrijitor.react() = "$name din afara clasei se joaca cu ${this@Ingrijitor.name}" //neglijat }
open class Vet(name: String): Ingrijitor(name) {   override fun Felina.react() = "fuge de $name"   }
```

Dispatch recv

```
fun main() {
   val om = Ingrijitor("ingrijitorul")
   val pisica = Pisica()
   val maimutaMare = Primata("maimuta")
   val femeie = Vet("corin")
   mesajScris(Primata("alex"))//apel functie din primata
   mesajScris(MaimutaMare("gibon"))//apel functie din primata
   om.areGrija(pisica)//apel functie din felina
   om.areGrija(maimutaMare)//apel dispatcher intern
   listOf(om, femeie).forEach { ingrijitor ->
      println("${ingrijitor.javaClass.simpleName} ${ingrijitor.name}")//Vet corin
      ingrijitor.areGrija(pisica)//Felina face: fuge de corin
      ingrijitor.areGrija(maimutaMare)//Primata spune: maimuta se joaca cu
corin      }
}
```

Funcții de extensie - conflict posibil de nume

```
class Sclav {
   fun munca() = "*munca la birt*"
   private fun odihna() = "*odihna la vecina*"
}

fun Sclav.munca() = "munca cu mila" //neglijata
fun <T> Sclav.munca(t:T) = "*muncesc azi? nuuuu $t*"
fun Sclav.odihna() = "odihna in sant"

fun main()
{
   val sclav = Sclav()
   println(sclav.munca()) //apel f membru
   println(sclav.munca("de maine ma apuc"))
   println(sclav.odihna()) //apel f extensie
}
```

Funcții de extensie pentru obiecte

```
object Constructor {
}

fun Constructor.casaNouaCaramida() = "casa pe pamant"

class Proiectant {
   companion object {
   }
   object Birou {
   }
}
fun Proiectant.Companion.prototipNou() = "montaj test"
fun Proiectant.Birou.mapaDeLucrari() = listOf("Proiect casa", "Proiect bloc")
fun main()
{
   println(Constructor.casaNouaCaramida())
   println(Proiectant.prototipNou())
   Proiectant.Birou.mapaDeLucrari().forEach(::println)
}
```

Funcții de extensie pentru liste

```
fun <T> List<T>.tail(): List<T> = this.drop(1)

infix fun <T> T.prependTo(list: List<T>): List<T> = listOf(this) + list

fun <T> List<T>.destructured(): Pair<T, List<T>> = first() to tail()

fun main()
{
   val intregi = listOf(11, 5, 3, 8, 1, 9, 6, 2)
   println(intregi.tail())
   println(intregi.prependTo(intregi))
   println(intregi.destructured())
}
```

Funcții infix

```
infix fun Int.sumaSmecheraCu(i: Int) = this + i //exemplul 1
fun main() {
   println(4 sumaSmecheraCu  7)
   println(2.sumaSmecheraCu(11))
}

object Toate {  // exemplul 2
   infix fun aleTale(base: Pair<Masini, Noua>) {}
}
object Masini {
   infix fun ne(apartin: Apartin) = this
}
object Apartin
object Noua
fun main() {
   println(Toate aleTale  (Masini ne Apartin to Noua))
}
```

Funcția map

```
fun main(args: Array<String>) {
    val lista1 = listOf<Int>(7,15,24,19,8,45,65,55)
    val lista2 = List(10) {
        (1..12).shuffled().first()
    }
    val lista3 = (1..15).map { it }
    val transformareLista = lista1.map { it*2 }
    println("lista 1 =${lista1}")
    println("Transformare lista1 cu map -> $transformareLista")
    println("lista 2 =${lista2}")
    println("lista 3 =${lista3}")
}
```

si rezultatul executieil

```
ista 1 =[7, 15, 24, 19, 8, 45, 65, 55]
Transformare lista1 cu map -> [14, 30, 48, 38, 16, 90, 130, 110]
lista 2 =[11, 4, 1, 9, 7, 12, 10, 7, 2, 2]
lista 3 =[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Process finished with exit code 0
```

Funcția Filtru - filter

```
import kotlin.math.*

fun main(args: Array<String>) {
    val lista = 1.until(15).toList()
    val lista1 = (1..15).map { it }
    val sublistaImpare = lista.filter { it%2==0 }
    println("Sublista care contine numai numerele pare este -> $sublistaImpare")
    val listaPatrate = lista1.filter {
        val radacinaPatrata = sqrt(it.toDouble()).roundToInt()
        radacinaPatrata*radacinaPatrata==it //conditia de filtrare
    }
    println("filteredListPSquare -> $listaPatrate")
}
```

si rezultatul executiei

```
Sublista care contine numai numerele pare este -> [2, 4, 6, 8, 10, 12, 14]
filteredListPSquare -> [1, 4, 9]

Process finished with exit code 0
```

Funcția FlatMap

```
fun main(args: Array<String>) {
    val lista1 = List(10) {
        (26..120).shuffled().first()
    }
    val listaBatutaCuCiocanul = lista1.flatMap {
        it.rangeTo(it+2*it).toList()
    }
    println("Lista dupa aplicarea flat Map este $listaBatutaCuCiocanul")
}
```

si rezultatul executiei

```
lista de intrare este [80, 35, 76, 85, 58, 42, 43, 98, 97, 110]
Lista dupa aplicarea flat Map este [80, 81, 82, 83, 84, 85, 86, 87, 88, ... mai sunt destule elemente

Process finished with exit code 0
```

Funcții pentru tăiere submulțimi - drop

```
fun main(args: Array<String>) {
    val dimLista = 20
    val lista1 = listOf(2, 1, 1, 7, 6, -8, 9, -12)
    val lista = 1.until(dimLista).toList()
    val catTaiStanga = 8
    val catTaiDreapta = 9
    val taiDinStanga = lista.drop(catTaiStanga)
    val taiDinDreapta = lista.dropLast(catTaiDreapta)
    var extragSubinterval = lista.drop(catTaiStanga).dropLast(catTaiDreapta)
    var extragereSelectiva =lista1.dropWhile { e->e>0}
    println("lista originala $lista")
    println("elimin subinterval stang [1..${catTaiStanga}]) -> ${taiDinStanga}")
    println("elimin subinterval drept [dim lista - ${catTaiDreapta}...dim  lista] -> ${taiDinDreapta}")
    println("Extragere submultime de la ${catTaiStanga} la ${catTaiDreapta} ->${extragSubinterval}")
    println("Extragere selectiva pe baza predicat - $extragereSelectiva")
}
```

Funcții pentru extragere din submulțimi - take

- fun main(args: Array<String>) {
  val lista = 1.until(25).toList()
  val catlau = 12
  println("lau primele ${catlau} din lista -> ${lista.take(catlau)}")
  println("lau ultimile ${catlau} din lista -> ${lista.takeLast(catlau)}")
  println("lau primele ${catlau} din lista -> ${lista.takeWhile { it<=catlau }}")
  println("lau toate elem incepand de la indexul ${catlau} -> ${lista.takeLastWhile { it>=catlau }}")
}

Funcția Fermoar - Zip

```
fun main(args: Array<String>) {
    val lista1 = listOf(1,2,3,4,5)
    val lista2 = listOf(
        "element 1",
        "element 2",
        "element 3",
        "element 4",
        "element 5"
    )

    val listaRezultat = lista1.zip(lista2)
    println("Utilizare zipWithNext -> ${lista1.zipWithNext()}")
    println(listaRezultat)
}
```

si rezultatul executiei

```
Utilizare zipWithNext -> [(1, 2), (2, 3), (3, 4), (4, 5)]
[(1, element 1), (2, element 2), (3, element 3), (4, element
4), (5, element 5)]

Process finished with exit code 0
```

Gruparea colecțiilor

```
fun main(args: Array<String>) {
    val lista = 1.rangeTo(20).toList()
    println(lista.groupBy { it%3 })
    println(lista.groupingBy { it }.eachCount())
}
```

si rezultatul executiei

```
{1=[1, 4, 7, 10, 13, 16, 19], 2=[2, 5, 8, 11, 14, 17, 20], 0=[3, 6, 9, 12, 15, 18]}
{1=1, 2=1, 3=1, 4=1, 5=1, 6=1, 7=1, 8=1, 9=1, 10=1, 11=1, 12=1, 13=1, 14=1, 15=1, 16=1, 17=1, 18=1, 19=1, 20=1}

Process finished with exit code 0
```

Functori - Functors

```
fun main(args: Array<String>) {
    listOf(1, 2, 3)
            .map { i -> i * 2 }
            .map(Int::toString)
            .forEach(::println)
}
```

Functori

```
sealed class TipSimplu<out T> {
    object None : TipSimplu<Nothing>() {
        override fun toString() = "cu None"
    }
    data class Some<out T>(val value: T) : TipSimplu<T>()
    companion object
}
fun <T, R> TipSimplu<T>.map(transform: (T) -> R): TipSimplu<R> = when (this) {
    TipSimplu.None -> TipSimplu.None
    is TipSimplu.Some -> TipSimplu.Some(transform(value))
}
fun main(args: Array<String>) {
    println(TipSimplu.Some("Simulare cu Some")
        .map(String::toUpperCase))
}
```

si rezultatul executiei

```
Some(value=SIMULARE CU SOME)

Process finished with exit code 0
```

Functori

si rezultatul executiei

```
Some(value=SIMULARE CU SOME)
None

Process finished with exit code 0
```

```
sealed class TipSimplu<out T> {
    object None : TipSimplu<Nothing>() {
        override fun toString() = "None"
    }
    data class Some<out T>(val value: T) : TipSimplu<T>()
    companion object
}
fun <T, R> TipSimplu<T>.map(transform: (T) -> R): TipSimplu<R> = when (this) {
    TipSimplu.None -> TipSimplu.None
    is TipSimplu.Some -> TipSimplu.Some(transform(value))
}
fun main(args: Array<String>) {
    println(TipSimplu.Some("Simulare cu Some").map(String::toUpperCase))
    println(TipSimplu.None.map(String::toUpperCase))
}
```

Functori

Pornim de la:

```
fun <A, B, C> ((A) -> B).map(transformare: (B) -> C): (A) -> C = { t -> transformare(this(t)) }
```

- Și avem două posibile implementări

```
//versiunea 1
typealias FunctieCuInt = (Int) -> Int
infix fun FunctieCuInt.map(g: FunctieCuInt): FunctieCuInt {
    return { x -> this(g(x)) }
}
val aduna3SiInmultesteCu3 = { a: Int -> a + 2 } map { a: Int -> a * 3 }
//versiunea 2
val aduna5SiInmultesteCu5: (Int) -> Int = { i: Int -> i + 3 }.map { j -> j * 2 }
fun main(args: Array<String>) {
    println(aduna3SiInmultesteCu3(33))
    println(aduna5SiInmultesteCu5(66))
}
```

Functori cu liste

```
fun List<Int>.convertLaStr(): List<String> =
   if (size > 0) {
      val ListaNoua = ArrayList<String>(size)
      for (element in this) {
         ListaNoua.add(procesare(procesareLambda(element,5)).toString())
      }
      ListaNoua
   } else {
      emptyList()
   }
val  procesareLambda = { x: Int, y: Int -> x - y }
fun procesare(x:Int):Int
{
   return x+5
}
fun main(args: Array<String>) {
   val ListaIntregi = listOf(271, 3, 17, 23, 51)
   println(ListaIntregi.convertLaStr())
}
```

Functor cu arbore

```
class ArboreBinar<T>(var value: T) {
    var stanga : ArboreBinar<T>? = null
    var dreapta: ArboreBinar<T>? = null
    fun <U> map(f: (T) -> U): ArboreBinar<U> {
        val arbore = ArboreBinar<U>(f(value))
        if (stanga  != null) arbore.stanga  = stanga?.map(f)
        if (dreapta != null) arbore.dreapta = dreapta?.map(f)
        return arbore
    }
    fun AfisezParteaSuperioara() = "(${stanga?.value}, $value, ${dreapta?.value})"
}
fun main(args: Array<String>) {
    val barb   = ArboreBinar(6)
    barb.stanga  = ArboreBinar(20)
    barb.dreapta = ArboreBinar(9)
    println(barb.AfisezParteaSuperioara())
    val barb1  = barb.map { it * 50.0 }
    println(barb1.AfisezParteaSuperioara())
}
```

si exemplu de executie

```
(20, 6, 9)
(1000.0, 300.0, 450.0)

Process finished with exit code 0
```

Monade

```
fun main(args: Array<String>) {
    val rezultat = listOf(1, 2, 3)
        .flatMap { i ->
            listOf(i * 2, i + 3)
        }
        .joinToString()

    println("La procesarea monadica a rezultat${rezultat}")
}
```

si rezultatul executiei

```
La procesarea monadica a rezultat2, 4, 4, 5, 6, 6

Process finished with exit code 0
```

Monade

```
sealed class TipSimplu<out T> {
    object None : TipSimplu<Nothing>() {
        override fun toString() = "None"
    }
    data class Some<out T>(val value: T) : TipSimplu<T>()
    companion object
}
fun <T, R> TipSimplu<T>.flatMap(fm: (T) -> TipSimplu<R>): TipSimplu<R> = when (this) {
    TipSimplu.None -> TipSimplu.None
    is TipSimplu.Some -> fm(value)
}
fun calculReducere(pret: TipSimplu<Double>): TipSimplu<Double> {
    return pret.flatMap { p ->
        if (p > 50.0) {
            TipSimplu.Some(5.0)
        } else {
            TipSimplu.None
        }
    }
}
fun main(args: Array<String>) {
    println(calculReducere(TipSimplu.Some(95.0)))
    println(calculReducere(TipSimplu.Some(25.0)))
    println(calculReducere(TipSimplu.None))
}
```

si rezultatul executiei

```
Some(value=5.0)
None
None

Process finished with exit code 0
```

Monade

```
sealed class TipSimplu<out T> {
   object None : TipSimplu<Nothing>() {
      override fun toString() = "None"
   }
   data class Some<out T>(val value: T) : TipSimplu<T>()
   companion object
}
fun <T, R> TipSimplu<T>.flatMap(fm: (T) -> TipSimplu<R>): TipSimplu<R> = when (this) {
   TipSimplu.None -> TipSimplu.None
   is TipSimplu.Some -> fm(value)
}
fun main(args: Array<String>) {
   val poatePatru = TipSimplu.Some(4)
   val poateSapte = TipSimplu.Some(7)
   println(poatePatru.flatMap { f ->
       poateSapte.flatMap { t ->
         TipSimplu.Some(f + t)
       }
   })
}
```

Si rezultatul executiei

```
Some(value=11)

Process finished with exit code 0
```

Monade

```
sealed class TipSimplu<out T> {
   object None : TipSimplu<Nothing>() {
      override fun toString() = "None"   }
   data class Some<out T>(val value: T) : TipSimplu<T>()
   companion object }
fun <T, R> TipSimplu<T>.flatMap(fm: (T) -> TipSimplu<R>): TipSimplu<R> = when (this) {
   TipSimplu.None -> TipSimplu.None
   is TipSimplu.Some -> fm(value)  }
fun main(args: Array<String>) {
   println(TipSimplu.Some(13).flatMap(::laJumatate))
   println(TipSimplu.Some(22).flatMap(::laJumatate))
   println(TipSimplu.None.flatMap(::laJumatate))
   TipSimplu.Some(34).flatMap(::laJumatate).flatMap( ::laJumatate).flatMap(::impOriDoi) }
fun laJumatate(a: Int) = when {
   a % 2 == 0 -> TipSimplu.Some(a / 2)
   else -> TipSimplu.None }
fun impOriDoi(a: Int) = when {
   a % 2 == 1 -> TipSimplu.Some(a * 2)
   else -> TipSimplu.None  }
```

si executia

```
None
Some(value=11)
None

Process finished with exit code 0
```

Applicatives

```
fun <T, R> List<T>.ap(fab: List<(T) -> R>): List<R> = fab.flatMap { f ->
this.map(f) }

fun main(args: Array<String>) {
    val numere = listOf(8, 13, 21, 34)
    val functii = listOf<(Int) -> Int>({ i -> i * 2 }, { i -> i + 3 })
    val rezultat = numere
        .ap(functii)
        .joinToString()
    println(rezultat)
}
```

Cum se poate face o funcție să se comporte ca un aplicative?

```
object functie {
   fun <A, B> pura(b: B) = { _: A -> b } }
fun <A, B, C> ((A) -> B).map(transform: (B) -> C): (A) -> C = { t -> transform(this(t)) }
fun <A, B, C> ((A) -> B).flatMap(fm: (B) -> (A) -> C): (A) -> C = { t -> fm(this(t))(t) }
fun <A, B, C> ((A) -> B).ap(fab: (A) -> (B) -> C): (A) -> C = fab.flatMap { f -> map(f) }
fun main(args: Array<String>) {
   val sumCu7SiMulCu11: (Int) -> Int = { i: Int -> i + 7 }.ap { { j: Int -> j * 11 } }
   println(sumCu7SiMulCu11(7))
   println(sumCu7SiMulCu11(8))
   println(sumCu7SiMulCu11(9))
   val sumCu7SiMulCu11Deb: (Int) -> Pair<Int, Int> = { i:Int -> i + 7 }.ap { original ->
{ j:Int -> original to (j * 11) } }
   println(sumCu7SiMulCu11Deb(10))
   println(sumCu7SiMulCu11Deb(11))
   println(sumCu7SiMulCu11Deb(12))
}
```

```
154
165
176
(10, 187)
(11, 198)
(12, 209)
Process finished with exit code 0
```

Aplicative

si executia

Some(value=11)

Process finished with exit code 0

```
sealed class TipSimplu<out T> {
   object None : TipSimplu<Nothing>() {
      override fun toString() = "None"   }
   data class Some<out T>(val value: T) : TipSimplu<T>()
   companion object }
fun <T, R> TipSimplu<T>.flatMap(fm: (T) -> TipSimplu<R>): TipSimplu<R> = when (this) {
   TipSimplu.None -> TipSimplu.None
   is TipSimplu.Some -> fm(value)  }
fun <T, R> TipSimplu<T>.map(transform: (T) -> R): TipSimplu<R> = when (this) {
   TipSimplu.None -> TipSimplu.None
   is TipSimplu.Some -> TipSimplu.Some(transform(value)) }
fun <T> TipSimplu.Companion.pur(t: T): TipSimplu<T> = TipSimplu.Some(t)
fun <T, R> List<T>.ap(fab: List<(T) -> R>): List<R> = fab.flatMap { f -> this.map(f) }
infix fun <T, R> TipSimplu<(T) -> R>.aplic (o: TipSimplu<T>): TipSimplu<R> = flatMap { f: (T) -> R ->
o.map(f) }
fun main(args: Array<String>) {
   val poatePatru = TipSimplu.Some(4)
   val poateSapte = TipSimplu.Some(7)
   println(TipSimplu.pur { f: Int -> { t: Int -> f + t } } aplic poatePatru aplic poateSapte) }
```

Aplicative

```
infix inline fun <A, reified B> Array<(A) -> B>.aplicatAsupraLui(a: Array<A>) =
    Array(this.size * a.size) {
        this[it / a.size](a[it % a.size])
    }


fun main(args: Array<String>) {
    val tablou = arrayOf<(Int) -> Int>({ it + 3 }, { it * 2 }) aplicatAsupraLui arrayOf(1,
2, 3)
    println("[ ${tablou.joinToString()} ]")
}
```

Currying vs aplicare parțială

	Currying	Aplicare parțială
Valoare întoarsă	când se primesc N parametri în funcție vor rezulta un lanț de funcții	cân se primesc n funcții rezultă o funcție primară și una care se aplică asupra celorlalte n-1
Utilizare parametri	după aplicare numai un parametru din lanț poate fi aplicat	orice parametru poate fi aplicat în orice ordine
inversare proces	poate reconveritită la o funcție cu N parametri	nu este posibil

Exemplu de curry/uncurry

```
fun <P1, P2, P3, R> Function3<P1, P2, P3, R>.curried(): (P1) -> (P2) -> (P3) -> R =
   { p1 -> { p2 ->   { p3 ->   this(p1, p2, p3)    }      }      }
fun <P1, P2, P3, R> ((P1) -> (P2) -> (P3) -> R).uncurried(): (P1, P2, P3) -> R {
   return { f1: P1, f2: P2, f3: P3 -> this(f1)(f2)(f3) }
}
var add = { x: Int, y: Int, z: Int-> x + y + z}.curried()
val add1 = { x: Int-> {y: Int -> {z: Int -> x+y+z}}}.uncurried()
fun main()
{
   val x = add(3)(4)(5)
   val y = add1(3,4,5)
   println(x)
   println(y)
}
```

Delegați pentru proprietăți (delegați standard)

- funcția Delegates.notNull și lateinit
- funcția lazy
- funcția Delegates.Observable
- funcția Delegates.vetoble

Delegates.notNull

```
import kotlin.properties.Delegates
class Loser {
    var nume: String by Delegates.notNull()
    fun initializare(nume: String) {
        this.nume = nume
    }
}
fun main() {
    val utilizator = Loser()
    utilizator.initializare("Bibistrocel")
    println(utilizator.nume)
}
```

Delegat construit de programator

```
import kotlin.reflect.KProperty
class ExempluDelegatPropriu {
    var valoareDelegata: String by DelegatPropriu()
    override fun toString() = "Exemplu delegat propriu"
}
class DelegatPropriu() {
    operator fun getValue(thisRef: Any?, prop: KProperty<*>): String {
        return "$thisRef am primit urmatoarea delegare '${prop.name}'"      }
    operator fun setValue(thisRef: Any?, prop: KProperty<*>, value: String) {
        println("$value a fost trimisa catre ${prop.name} in $thisRef")     }
}
fun main() {
    val exemplu = ExempluDelegatPropriu()
    println(exemplu.valoareDelegata)
    exemplu.valoareDelegata = "Unul nou"
}
```

lateinit

```
class SubiectTestare
{ fun zice()
   {      println("Iar ma chinuie astia")    }
}
public class PrimulTest {
   lateinit var subiect: SubiectTestare
   fun setup() {
      subiect = SubiectTestare()      }
   fun test() {
      if (::subiect.isInitialized) subiect.zice()      }
}
fun main()
{  val x=PrimulTest()
   x.setup() //comentati linia si vedeti ce se intampla
   x.test()
}
```

Lazy

```
fun main() {
    val i by lazy {
        println("Evaluare la cerere")
        1
    }
     println("inainte de utilizarea lui i")
    println(i)
}
```

Observable

```
import kotlin.properties.Delegates
class Looser {
    var nume: String by Delegates.observable("inca nu am nume") {
            proprietate, valoareVeche, valoareNoua ->
        println("$valoareVeche - $valoareNoua")
    }
}
fun main() {
    val utilizator = Looser()
    utilizator.nume = "Cel mai prost din curtea scolii"
    utilizator.nume = "Inca si mai prost"
    utilizator.nume = "bun de avansare"
}
```

Vetoable

```
import kotlin.properties.Delegates
class Looser {
    var valoareCrescatoare: Int by Delegates.vetoable(0) {
            proprietate, valoareVeche, valoareNoua ->
        if(valoareVeche > valoareNoua) true
        else throw IllegalArgumentException("Noua valoare trebuie sa fie mai mare decat cea veche")
    }
}
fun main()
{
    var x= Looser()
    x.valoareCrescatoare =10
    x.valoareCrescatoare = 12
    try{
        x.valoareCrescatoare =6
    } catch (e: java.lang.IllegalArgumentException) {
        println("ai un prietene in vizita prin program?")
    }
    x.valoareCrescatoare =15
    println(x.valoareCrescatoare)
}
```

Delegated map

```
data class Carte (val delegate:Map<String,Any?>) {
   val nume:String by delegate
   val autori:String by delegate
   val numarPagini:Int by delegate
   val dataPublicarii:String by delegate
   val editura:String by delegate  }
fun main() {
   val mapCarte1 = mapOf(
      Pair("nume","Povesti corecte politic de adormit copiii"),
      Pair("autori","James Finn Garner"),
      Pair("pageCount",200),
      Pair("dataPublicarii","01/06/2006"),
      Pair("editura","Humanitas")      )
   val mapCarte2 = mapOf(
      "nume" to "Critica ratiunii pure",
      "autori" to "Immanuel Kant",
      "numarPagini" to 250,
      "dataPublicarii" to "12/05/1998",
      "editura" to "IRI"      )
   val cartea1 = Carte(mapCarte1)
   val cartea2 = Carte(mapCarte2)
   println("Prima Carte \n$cartea1  \nA doua Carte \n$cartea2")
}
```

Delegați pentru clase

```
interface Persoana {
    fun afiseazaNume()
}
class ImplementarePersoana(val nume:String):Persoana {
    override fun afiseazaNume() {
        println(nume)
    }
}
class Utilizator(val persoana:Persoana):Persoana by persoana {
    override fun afiseazaNume() {
        println("Numele este:")
        persoana.afiseazaNume()
    }
}
fun main() {
    val persoana = ImplementarePersoana("Bugs Bunny")
    persoana.afiseazaNume()
    val utilizator = Utilizator(persoana)
    utilizator.afiseazaNume()
}
```