Programare declarativă

Introducere în programarea funcțională folosind Haskell
- recapitulare -

Traian Florin Șerbănuță - seria 33
Ioana Leuștean - seria 34

Departamentul de Informatică, FMI, UB
traian.serbanuta@fmi.unibuc.ro
ioana@fmi.unibuc.ro

Programarea funcțională

O cale profund diferită de a concepe ideea de software

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări
- Distincție clară între părțile pure și cele care comunică cu mediul extern
- Haskell e leneș: orice calcul e amânat cât de mult posibil
  - Schimbă modul de concepere al programelor
  - Permite lucrul cu colecții potențial infinite de date precum [1..]

Limbajul Haskell

- Puritatea asigură consistență
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul
- Evaluarea lenesă poate fi exploatată pentru a reduce timpul de calcul fără a denatura codul

```
firstK k xs = take k (sort xs)
```

- Minimalism: mai puțin cod, în mai puțin timp, și cu mai puține defecte
  - ...rezolvând totuși problema :-)

```
numbers = [1,2,3,4,5]
total = foldl (*) 0 numbers
doubled = map (* 2) numbers
```

- Oferă suport pentru paralelism și concurență

Legarea variabilelor

= nu este atribuire

În Haskell, variabilele sunt imuabile, adică:

- = **nu** este operator de atribuire
- x = 1 reprezintă o *legatură* (binding)
- din momentul în care o variabilă este legată la o valoare, acea valoare nu mai poate fi schimbată

În Haskell, evaluarea este leneșă, adică expresiile sunt evaluate numai atunci când este nevoie!

```
Prelude> x=1
Prelude> x = x+1 -- nu se produce mesaj de eroare
Prelude> x -- se evalueaza x, evaluarea nu se termina
```

let și where

let .. in ...

este o *expresie* care crează scop local

```
Prelude> x=1
Prelude> z= let x=3 in x
Prelude> z
3
Prelude> x
1
```

.. where ...

este o *clauză* care crează scop local

```
Prelude> f x = g (x+1) where g x = x*x
Prelude> f 3
16
Prelude> g 3
error
```

Legarea variabilelor

- **let .. in ...** este o expresie

```
x = [let y =8 in y, 9] -- x=[8,9]
```

- **where** este o clauză, disponibilă doar la nivel de definiție

x = [y where y =8, 9] – error: parse error ...

- Variabile pot fi legate și prin "pattern matching" la definirea unei funcții sau expresii **case**.

```
h x | x == 0    = 0
    | x == 1    = y + 1
    | x == 2    = y * y
    | otherwise = y
  where y = x*x
```

```
f x = case x of
        0 -> 0
        1 -> y + 1
        2 -> y * y
        _ -> y
  where y = x*x
```

Tipuri și clase de tipuri

Sistemul tipurilor

"There are three interesting aspects to types in Haskell: they are strong, they are static, and they can be automatically inferred."
http://book.realworldhaskell.org/read/types-and-functions.html

tare garanteaza absenta anumitor erori

static tipul fiecari valori este calculat la compilare

dedus automat compilatorul deduce automat tipul fiecarei expresii

```
Prelude> :t [('a',1,"abc")]
[('a',1,"abc")] :: Num b => [(Char, b, [Char])]
```

Sistemul tipurilor

Tipurile de baza

Int, Integer, Float, Double, Bool, Char, String

- tipuri compuse: tupluri si liste

```
Prelude> :t :t ('a', True) -- tuplurile pot contine
    elemente cu
('a', True) :: (Char, Bool) -- tipuri diferite

Prelude> :t ["ana", "ion"] -- listele contin elemente
["ana", "ion"] :: [[Char]] -- de acelasi tip
```

- tipuri noi definite de utilizator

```
data MyList a = Nil | Cons a (MyList a) -- tipuri
    algebrice
newtype MyVar = V String -- un singur constructor
type Nume = String -- redenumiri de tipuri
```

Tipuri de bază

- Integer: 4, 0, -5

```
Prelude> 4 + 3
Prelude> (+) 4 3
```

```
Prelude> mod 4 3
Prelude> 4 `mod` 3
```

- Float: 3.14

```
Prelude> truncate 3.14
Prelude> sqrt 4
```

```
Prelude> let x = 4 :: Int
Prelude> sqrt (fromIntegral x)
```

- Char: 'a','A', '\n'

```
Prelude> import Data.Char
Prelude Data.Char> chr 65
Prelude Data.Char> ord 'A'
Prelude Data.Char> toUpper 'a'
Prelude Data.Char> digitToInt '4'
```

Tipuri de bază

- Bool: True, False

```
data Bool = True | False
```

```
Prelude> True && False || True
Prelude> not True
```

```
Prelude> 1 /= 2
Prelude> 1 == 2
```

- String: "prog\ndec"

```
type String = [Char] -- sinonim pentru tip
```

```
Prelude> "aa"++"bb"
"aabb"
Prelude> "aabb" !! 2
'b'
```

```
Prelude> lines "prog\ndec"
["prog","dec"]
Prelude> words "pr og\nde cl"
["pr","og","de","cl"]
```

Clase de tipuri. Variabile de tip

Ce răspuns primim in GHCi dacă introducem comanda

```
Prelude> :t 1
```

Răspunsul primit este:

```
1 :: Num a => a
```

Semnificația este următoarea:

- a este un *parametru de tip*
- Num este o clasă de tipuri
- 1 este o valoare de tipul a din clasa Num

```
Prelude> :t 1
1 :: Num a => a
```

Putem defini tipuri parametrizate:

```
data Point a = Pt a a      -- tip parametrizat

Prelude> :t Pt "a" "b"
Pt "a" "a" :: Point [Char]
```

Clase de tipuri

O clasă de tipuri este o colecție de operații (este o interfață).
În clasa **Num** regăsim acele date care au definite operațiile de adunare, scădere, înmulțire, etc.

```
Prelude> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
instance Num Integer -- Defined in 'GHC.Num'
instance Num Int -- Defined in 'GHC.Num'
instance Num Float -- Defined in 'GHC.Float'
instance Num Double -- Defined in 'GHC.Float'
```

Clase de tipuri

- Clasa **Eq** conține tipurile care au definită relația de egalitate:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
  -- ^^^ putem avea definitii implicite
```

- Tipurile care aparțin clasei sunt instanțe ale clasei.

```
instance Eq Bool where
  False == False = True
  False == True  = False
  True  == False = False
  True  == True  = True
```

Clasa de tipuri. Constrângeri de tip

- Funcția **elem** verifică dacă un element aparține unei liste:

```
Prelude> elem 2 [1,2,3]
True
Prelude> elem "a" "curs"
Prelude> elem 'a' "curs"
False
```

- În signatura funcției **elem** trebuie să precizăm ca tipul a este în clasa **Eq**

```
elem :: Eq a => a -> [a] -> Bool
```

Eq a se numește constrângere de tip. **=>** separă constrăngerile de tip de restul signaturii.

- În exemplul de mai sus am considerat că **elem** este definită pe liste, dar în realitate funcția este mai complexă!

Definirea claselor

Să presupunem că vrem să definim o clasă de tipuri pentru datele care pot fi afișate.
O astfel de clasă trebuie să conțină o metodă care să indice modul de afișare:

```
class Visible a where
    toString :: a -> String
```

Putem face instanțieri astfel:

```
instance Visible Char where
    toString c = [c]
```

Clasele **Eq**, **Ord** sunt predefinite. Clasa Visible este definită de noi, dar există o clasă predefinită care are același rol: clasa **Show**

Show

```
class Show a where
  show :: a -> String     -- analogul lui "toString"

instance Show Bool where
  show False      = "False"
  show True       = "True"

instance (Show a, Show b) => Show (a,b) where
  show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"

  instance Show a => Show [a] where
    show []     = "[]"
    show (x:xs) = "[" ++ showSep x xs ++ "]"
      where
        showSep x []     = show x
        showSep x (y:ys) = show x ++ "," ++ showSep y ys
```

Tipuri de date compuse

- Tipul tuplu - secvențe de de tipuri deja existente

```
Prelude> :t (1 :: Int, 'a', "ab")
(1 :: Int, 'a', "ab") :: (Int, Char, [Char])

Prelude> fst (1,'a') -- numai pentru perechi
Prelude> snd (1,'a')
```

- Tipul unit

```
Prelude> :t ()
() :: ()
```

- Tipul listă

```
Prelude>:t [True, False, True]
[True, False, True] :: [Bool]
```

Liste și tupluri

Liste

Definiție

Observație

Orice listă poate fi scrisă folosind doar constructorul (:) și lista vidă []

- [1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []
- "abcd" == ['a','b','c','d'] == 'a' : ('b' : ('c' : ('d' : []))) == 'a' : 'b' : 'c' : 'd' : []

Definiție recursivă

O listă este

- vidă, notată []; sau
- compusă, notată x:xs, dintr-un un element x numit capul listei (head) și o listă xs numită coada listei (tail).

Definirea listelor. Operații

Intervale și progresii

```
interval = ['c'..'e']          -- ['c','d','e']
progresie = [20,17..1]         -- [20,17,14,11,8,5,2]
progresie' = [2.0,2.5..4.0]    -- [2.0,2.5,3.0,3.5,4.0]
```

Operații

```
Prelude> [1,2,3] !! 2
3
Prelude> "abcd" !! 0
'a'
Prelude> [1,2] ++ [3]
[1,2,3]
Prelude> import Data.List
```

Definiția prin selecție $\{x \mid P(x)\}$

[E(x)| x <- [x1,. . . ,xn], P(x)]

```
Prelude> let xs = [0..10]
Prelude> [x | x <- xs, even x]
[0,2,4,6,8,10]
```

```
Prelude> let xs = [0..6]
Prelude> [(x,y) | x <- xs, y <- xs,  x + y == 10]
[(4,6),(5,5),(6,4)]
```

Folosirea lui `let` pentru declarații locale:

```
Prelude> [(i,j) | i <- [1..2], let k = 2 * i, j <- [1..k]]
[(1,1),(1,2),(2,1),(2,2),(2,3),(2,4)]
```

Lenevire (Lazyness)

Argumentele sunt evaluate doar când e necesar și doar cât e necesar

```
Prelude> head[]
*** Exception: Prelude.head: empty list
Prelude> let x = head []
Prelude> let f a = 5
Prelude> f x
5
Prelude> [1,head [],3] !! 0
1
Prelude> [head [],3] !! 1
*** Exception: Prelude.head: empty list
```

Liste infinite

Drept consecință a evaluării lenești, se pot defini liste infinite (fluxuri de date)

```
Prelude> let natural = [0,..]
Prelude> take 5 natural
[0,1,2,3,4]

Prelude> let evenNat = [0,2..] -- progresie infinita
Prelude> take 7 evenNat
[0,2,4,6,8,10,12]

Prelude> let ones = [1,1..]
Prelude> let zeros = [0,0..]
Prelude> let both = zip ones zeros
Prelude> take 5 both
[(1,0),(1,0),(1,0),(1,0),(1,0)]
```

Tupluri

- **fst** și **snd**

```
Prelude> fst (1, True)
1
```

```
Prelude> snd (1, True)
True
```

- funcția **zip**

```
Prelude> zip [1 .. 4] ["one", "two", "three", "four"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four")]
```

- **zip** vs comprehensiune

```
Prelude> [(x,y)| x<-[1..3], y<-["one", "two", "three"]]
[(1,"one"),(1,"two"),(1,"three"),(2,"one"),(2,"two"),(2,
    "three"),(3,"one"),(3,"two"),(3,"three")]
```

Exerciții

- selectarea elementelor din pozitii pare folosind **zip**

```
Prelude> let xs = ['A'..'Z']
Prelude> [x | (i,x) <- [1..] `zip` xs, even i]
"BDFHJLNPRTVXZ"
```

- funcție care elimină elementul din poziția n, întorcând o pereche formată din element și lista rămasă

```
removeAt :: Int -> [a] -> (a, [a])
removeAt n xs = ((xs !! n),
                 [ x | (i, x) <- zip [0..] xs, i /= n ])

Prelude> removeAt 0 [1,2,3]
(1,[2,3])
Prelude> removeAt 4 [1,2,3]
*** Exception: Prelude.!!: index too large
```

Funcții și șabloane

Funcții în Haskell. Terminologie

Exemplu: adunarea a doi întregi

Prototipul funcției

add :: Integer -> Integer -> Integer

- **numele funcției**
- signatura funcției

Definiția funcției

add elem1 elem2 = elem1 + elem2

- **numele funcției**
- parametrii formali
- corpul funcției

Aplicarea funcției

add 3 7

- **numele funcției**
- argumentele

Funcții în Haskell. Terminologie

Exemplu: funcție cu **un** argument de tip tuplu

Prototipul funcției

dist :: (Integer, Integer) -> Integer

- **numele funcției**
- signatura funcției

Definiția funcției

dist (elem1, elem2) = abs (elem1 - elem2)

- **numele funcției**
- parametrul formal
- corpul funcției

Aplicarea funcției

dist (2, 5)

- **numele funcției**
- argumentul

Tipuri de funcții

Fie foo o funcție cu următorul tip

```
foo :: a -> b -> [a] -> [b]
```

- are trei argumente, de tipuri a, b și [a]
- întoarce un rezultat de tip [b]

Schimbăm signatura funcției astfel:

```
ffoo :: (a -> b) -> [a] -> [b]
```

- are două argumente, de tipuri (a -> b) și [a],
  adică o funcție de la a la b și o listă de elemente de tip a
- întoarce un rezultat de tip [b]

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

Operatorii sunt funcții cu două argumente

Operatorii în Haskell

- pot fi definiți folosind numai "simboluri" (ex: *!*)
- au două argumente
- sunt apelați folosind notația infix

- Operatori predefiniți

```
(||) :: Bool -> Bool -> Bool
(:) :: a -> [a] -> [a]
(+) :: Num a => a -> a -> a
```

- Operatori definiți de utilizator

```
(&&&) :: Bool -> Bool -> Bool  -- atentie la paranteze
True &&& b = b
False &&& _ = False
```

Precedență și asociativitate

```
Prelude> 3+5*4:[6]++8-2+3:[2]==[23,6,9,2]||True==False
True
```

Precedence	Left associative	Non-associative	Right associative
9	!!		.
8			^, ^^, **
7	*, /, ‘**div**‘, ‘**mod**‘, ‘**rem**‘, ‘**quot**‘		
6	+, −		
5			:, ++
4		==, /=, <, <=, >, >=, ‘**elem**‘, ‘**notElem**‘	
3			&&
2			||
1	>>, >>=		
0			$, $!, ‘**seq**‘

Secțiuni ("operator sections")

Secțiunile operatorului binar op sunt (op e) și (e op).
Matematic, ele corespund aplicării parțiale a funcției op.

- secțiunile lui || sunt (|| e) și (e ||)

```
Prelude> :t (|| True)
(|| True) :: Bool -> Bool
Prelude> (|| True) False  -- atentie la paranteze
True
Prelude> || True False
error
```

- secțiunile lui <+> sunt (<+> e) și (e <+>), unde

```
Prelude> let x <+> y = x+y+1  -- definit de utilizator
Prelude> :t (<+> 3)
(<+> 3) :: Num a => a -> a
Prelude> (<+> 3) 4
8
```

Secțiuni

- Secțiunile operatorului (:)

```
Prelude> (2:)[1,2]
[2,1,2]
Prelude> (:[1,2]) 3
[3,1,2]
Prelude>
```

- Secțiunile sunt afectate de asociativitatea și precedența operatorilor.

```
Prelude> :t (+ 3 * 4)
(+ 3 * 4) :: Num a => a -> a

Prelude> :t (* 3 + 4)
error -- + are precedenta mai mica decat *
Prelude> :t (* 3 * 4)
error  -- * este asociativa la stanga
Prelude> :t (3 * 4 *)
(3 * 4 *) :: Num a => a -> a
```

Funcții anonime și secțiuni

Funcții anonime = lambda expresii

\x1 x2 ··· xn -> expresie

```
Prelude> (\x -> x + 1) 3
4
Prelude> inc = \x -> x + 1
Prelude> add = \x y -> x + y
Prelude> aplic = \f x -> f x
```

Sectiunile sunt definite prin lambda expresii:

```
(x +) = \y -> x+y
(+ y) = \x -> x+y
```

Compunerea funcțiilor — operatorul .

Matematic

Date fiind $f : A \to B$ și $g : B \to C$, compunerea lor, notată $g \circ f : A \to C$ este dată de formula

$$(g \circ f)(x) = g(f(x))$$

În Haskell

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g . f) x = g (f x)
```

Exemplu

```
Prelude> z=1
Prelude> t=2
Prelude> sqrt(z^2+t^2)
2.23606797749979

Prelude> x = 1 :: Integer
Prelude> y = 3 :: Integer
Prelude> sqrt fromIntegral (x^2+y^2)
<interactive>:33:1: error:


Prelude> sqrt . fromIntegral (x^2+y^2)
<interactive>:36:1: error:@*

Prelude> (sqrt . fromIntegral) (x^2+y^2)
3.1622776601683795
```

Operatorul $

Operatorul ($) are precedența 0.

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

```
Prelude> sqrt 3 + 4 +9
14.732050807568877
Prelude> sqrt (3 + 4 +9)
4.0
Prelude> sqrt $ 3 + 4 +9
4.0
```

Operatorul ($) este asociativ la dreapta.

```
Prelude> sqrt $ fromIntegral $ x^2+y^2
3.162277660168379 5
```

Definirea funcțiilor folosind **if**

- analiza cazurilor folosind expresia "if"

```
semn : Integer -> Integer
semn n = if n < 0 then (-1)
         else if n=0 then 0
              else 1
```

- definiție recursivă în care analiza cazurilor folosește expresia "if"

```
fact :: Integer -> Integer
fact n = if n == 0 then 1
         else n * fact(n-1)
```

Definirea funcțiilor folosind **gărzi**

Funcția *semn* o putem defini astfel

$$\mathit{semn}\ n = \begin{cases} -1, & \textbf{dacă n} < \textbf{0} \\ 0, & \textbf{dacă n} = \textbf{0} \\ 1, & \textbf{altfel} \end{cases}$$

În Haskell, condițiile devin gărzi:

```
semn  n
      | n < 0          =  -1
      | n = 0          =   0
      | otherwise      =  1
```

Definirea funcțiilor folosind **gărzi**

Funcția *fact* o putem defini astfel

$$fact\ n = \begin{cases} 1, & \textbf{dacă n} = \textbf{0} \\ n * fact(n-1), & \textbf{altfel} \end{cases}$$

În Haskell, condițiile devin gărzi:

```
fact n
    | n == 0      = 1
    | otherwise   = n * fact(n-1)
```

Definirea funcțiilor folosind șabloane și ecuații

```
semn :: Integer -> Integer
semn 0 = 0
semn x
   | x > 0      = 1
   | otherwise = -1

fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact(n-1)
```

- variabilele și valorile din partea stângă a semnului = sunt *șabloane*;
- când funcția este aplelată se încearcă potrivarea parametrilor actuali cu șabloanele, ecuațiile fiind încercate *în ordinea scrierii*;
- în definiția factorialului, 0 și n sunt șabloane: 0 se va potrivi numai cu el însuși, iar n se va potrivi cu orice valoare de tip `Integer`.

Definirea funcțiilor folosind șabloane și ecuații

- în Haskell, ordinea ecuațiilor este importantă

Să presupunem că schimbăm ordinii ecuațiilor din definiția factorialului:

```
fact :: Integer -> Integer
fact n = n * fact(n-1)
fact 0 = 1
```

Ce se întâmplă?

Deoarece n este un pattern care se potrivește cu orice valoare, inclusiv cu 0, orice apel al funcției va alege prima ecuație. Astfel, funcția nu își va încheia execuția pentru valori pozitive.

Definirea funcțiilor folosind șabloane și ecuații

Tipul Bool este definit în Haskell astfel:

```
data Bool = True | False
```

Putem defini operația || astfel

```
(||) :: Bool -> Bool -> Bool

False || x = x
True  || _ = True
```

În acest exemplu șabloanele sunt _, **True** și **False**.

Observăm că **True** și **False** sunt constructori de date și se vor potrivi numai cu ei înșiși.

Șablonul _ se numește *wild-card pattern*; el se potrivește cu orice valoare.

Șabloane (patterns) pentru liste

Listele sunt construite folosind constructorii (:) și []

```
[1,2,3] == 1:[2,3]  -- == 1:2:[3] == 1:2:3:[]
```

Observați:

```
Prelude> let x:y = [1,2,3]
Prelude> x
1
Prelude> y
[2,3]
```

Ce s-a întâmplat?

- x:y este un șablon pentru liste
- potrivirea dintre x:y și [1,2,3] a avut ca efect:
  - "deconstrucția" valorii [1,2,3] în 1:[2,3]
  - legarea lui x la 1 și a lui y la [2,3]

Șabloane (patterns) pentru liste

Definiții folosind șabloane

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

- x:xs se potrivește cu liste nevide

Atenție!

Șabloanele sunt definite folosind constructori. De exemplu, operația de concatenare pe liste este (++) :: [a]-> [a] -> [a] dar
[x] ++ [1] = [2,1] nu va avea ca efect legarea lui x la 2;
încercând să evaluăm x vom obține un mesaj de eroare:

```
Prelude> [x] ++ [1] = [2,1]
Prelude> x
<interactive>:83:1: error: ...
```

Șabloane pentru tupluri

Observați că (,) este constructorul pentru perechi.

```
(u,v)=('a',[(1,'a'),(2,'b')]) -- u='a',
                              -- v=[(1,'a'),(2,'b')]
```

- Definitii folosind șabloane

```
selectie :: Integer -> String -> String
```

```
-- case...of
selectie x s =
    case (x,s) of
        (0,_) -> s
        (1, z:zs) -> zs
        (1, []) -> []
        _ -> (s ++ s)
```

```
-- stil ecuational
selectie 0 s = s
selectie 1 (_:s) = s
selectie 1 "" = ""
selectie _ s = s + s
```

Șabloanele sunt liniare

În Haskell șabloanele sunt *liniare*, adică o variabilă apare cel mult odată. Șabloane în care o variabilă apare de mai multe ori provoacă mesaje de eroare

```
x:x:[1] = [2,2,1]
ttail (x:x:t) = t
foo x x = x^2
```

Cum rezolvăm problema în astfel de situații?
O soluție este folosirea gărzilor:

```
ttail (x:y:t) | (x==y) = t
              | otherwise = ...

foo x y | (x == y) = x^2
        | otherwise = ...
```

Funcții de nivel înalt

Funcțiile sunt valori

Funcțiile — „cetățeni de rangul I”

Funcțiile sunt valori,
care pot fi trimise ca argument sau întoarse ca rezultat

Exemplu:

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

- definiția cu lambda expresii

```
flip f = \x y -> f y x
```

- definiția folosind șabloane

```
flip f x y = f y x
```

- **flip** ca valoare de tip funcție

```
flip = \f x y -> f y x
```

Funcții de ordin înalt

map

```
map :: (a -> b) -> [a] -> [b]
map f l = [f x | x <- l]
```

```
Prelude> map (* 3) [1,3,4]
[3,9,12]
```

Un exemplu mai complicat:

```
Prelude> map ($ 3) [(4 +), (10 *), (^ 2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

În acest caz:

- primul argument este o sectiune a operatorului ($)
- al doilea argument este o lista de functii

  map ($ x) [f_1,. . ., f_n] == [f_1 x,. . ., f_n x]

Funcții de ordin înalt

filter și map

```
filter :: (a -> Bool) -> [a] -> [a]
filter p l = [x | x <- l, p x]

Prelude> filter (>= 2) [1,3,4]
[3,4]
```

Filtrare și aplicare

```
Prelude> let f l = map (* 3) (filter (>= 2) l)
Prelude> f [1,3,4]
[9, 12]                          -- [ x * 3 | x <- [1,3,4], x >=2 ]
```

Definiția compozițională (pointfree style)

```
f = map (* 3) . filter (>=2)
```

Funcții de ordin înalt

filter și map

http://learnyouahaskell.com/higher-order-functions

Calculați suma pătratelor impare mai mici decât 10000.

```
oddSquareSum :: Integer
oddSquareSum = let
                 squares = map (^2) [1..]
                 oddSquares = filter odd squares
                 belowLimit = takeWhile (<10000) oddSquares
               in  sum belowLimit
```

Observați funcția **takeWhile**

```
takeWhile :: (a -> Bool) -> [a] -> [a]
Prelude> takeWhile odd [1,3,5,4,7]
[1,3,5] -- cel mai lung prefix cu elemente impare
```

Definiția compozițională

```
oddSquareSum = sum . takeWhile (<10000) . filter odd .
                                          map (^2) $ [1..]
```

foldr și foldl

Definiție

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoare obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

Funcția *foldr*

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i []     =  i
foldr f i (x:xs) = f x (foldr f i xs)
```

Funcția *foldl*

```
foldl :: (b -> a-> b) -> b -> [a] -> b
foldl h i []     =  i
foldl h i (x:xs) = foldl h (h i  x) xs
```

Funcții de ordin înalt

foldr și foldl

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr o z [a1, a2, a3, ..., an] =
                        a1 `o` (a2 `o` (a3 `o` (... (an `o` z) ...)))
```

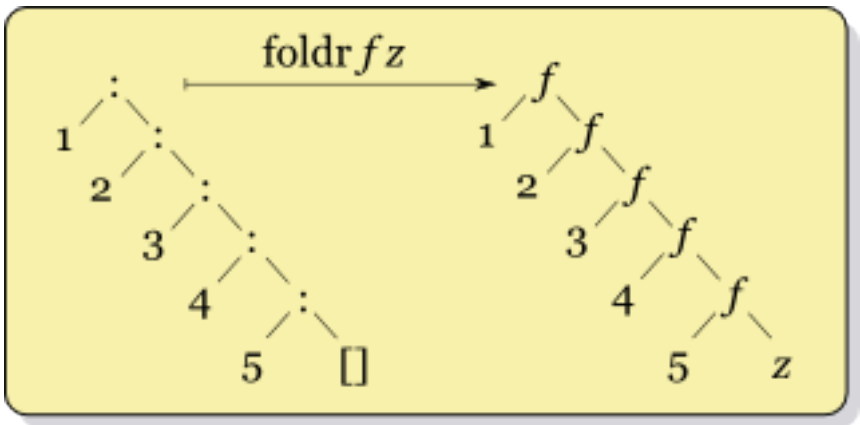
```
Prelude> foldr (*) 1 [1,3,4]
12                          -- product [1,3,4]
```

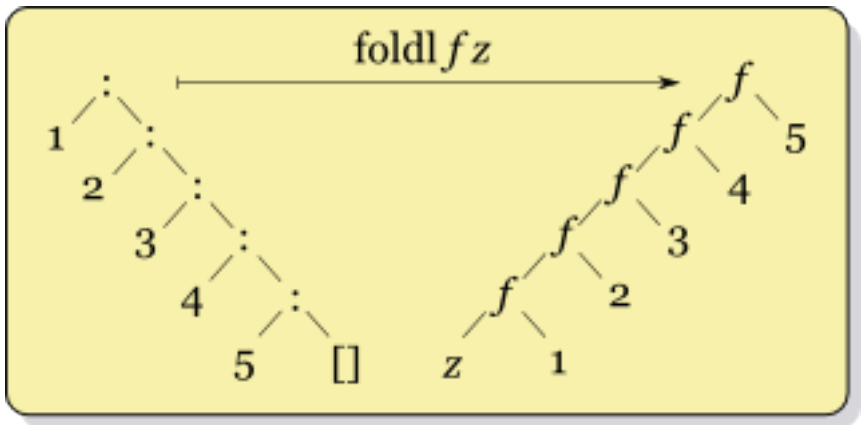
```
foldl :: (b -> a -> b) -> b -> [a] -> b

foldl o z [a1, a2, a3, ..., an] =
                        ( ... (((z `o` a1) `o` a2) `o` a3) `o` ... an)
```

```
Prelude> foldl (flip (:)) [] [1,3,4]
[4,3,1]   -- de ce? intelegeti modul de functionare!
```

foldr și foldl





https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Care dintre ele poate fi folosită pe liste infinite?

- **foldr** poate fi folosită pe liste infinite (în anumite cazuri),
- **foldl** nu poate fi folosită pe liste infinite niciodată.

foldr și foldl

- **foldr** poate fi folosită pe liste infinite (în anumite cazuri),
- **foldl** nu poate fi folosită pe liste infinite niciodată.

```
Prelude> foldr (*) 0 [1..]
*** Exception: stack overflow
```

```
Prelude> take 3 $ foldr (\x xs-> (x+1):xs) [] [1..]
[2,3,4]
-- foldr a functionat pe o lista infinita
```

```
Prelude> take 3 $ foldl (\xs x-> (x+1):xs) [] [1..]
-- expresia se evalueaza la infinit
```

Filtrare, transformare, agregare

Suma pătratelor elementelor pozitive

- Folosind descrieri de liste și funcții de agregare standard

```
f :: [Int] -> Int
f xs = sum [x * x | x <- xs, x > 0]
```

- Folosind funcții auxiliare

```
f xs = foldr (+) 0 (map sqr (filter pos xs))
  where
    sqr x = x * x
    pos x = x > 0
```

- Folosind funcții anonime

```
f :: [Int] -> Int
f xs = foldr (+) 0
    (map (\x -> x * x) (filter (\x -> x > 0) xs))
```

Filtrare, transformare, agregare

Suma pătratelor elementelor pozitive

- Folsind secțiuni și operatorul $ (parametru explicit)

```
f :: [Int] -> Int
f xs = foldr (+) 0 $ map (^ 2) $ filter (> 0) xs
```

- Definiție compozițională (pointfree style)

```
f :: [Int] -> Int
f = foldr (+) 0 . map (^ 2) . filter (> 0)
```

Proprietatea de universalitate

Observație

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i :: [a] -> b
```

Teoremă

Fie g o funcție care procesează liste finite. Atunci

$$\begin{array}{lcl} g\ [] & = & i \\ g\ (x : xs) & = & f\ x\ (g\ xs) \end{array} \quad \Leftrightarrow \quad g = \mathit{foldr}\ f\ i$$

Demonstrație:

$\Rightarrow$ Înlocuind $g = \mathit{foldr}\ f\ i$ se obține definiția lui **foldr**

$\Leftarrow$ Prin inducție dupa lungimea listei.

Teorema determină condiții necesare și suficiente pentru ca o funcție g care procesează liste să poată fi definită folosind **foldr**.

Generarea funcțiilor cu **foldr**

Compunerea funcțiilor

În signatura lui **foldr**

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

a si b pot fi de tip funcție.

a si b sunt (c->c)

```
compose :: [c -> c] -> (c -> c)
compose = foldr (.) id
```

```
Prelude> foldr (.) id [(+1), (^2)] 3
10

-- functia (foldr (.) id [(+1), (^2)]) aplicata lui 3
```

Generarea funcțiilor cu **foldr**

```
foldr :: (a -> b -> b) -> b -> [a] -> b

mysum xs = foldr (+) 0 xs
myor  xs = foldr (||) False xs


mylength xs = foldr (\a b -> 1 + b) 0 xs

myreverse xs = foldr (\a b -> b ++ [a]) [] xs


mymap f xs = foldr (\x ys -> (f x) : ys) [] xs

myfilter p xs = foldr f [] xs
                where
                f = \x ys -> if (p x) then (x : ys) else ys
```

Exercițiu

Scrieți o funcție care are la intrare un șir reprezentând o expresie în forma prefix și care întoarce valoarea expresiei.

```
*Main> solve "- 10 * 2 + 4 3"
-4
```

```
solve = head . foldr f [] . words
        where
            f "*" (x:y:ys) = (x * y):ys
            f "+" (x:y:ys) = (x + y):ys
            f "-" (x:y:ys) = (x - y):ys
            f number xs = (read number :: Int):xs
```

Tipuri de date algebrice

Tip sumă: anotimpuri

- tip sumă

```
data Season = Spring | Summer
            | Autumn | Winter
```

- tip produs

```
data Point a b = Pt a b
```

Observați că:

Point este constructor de tip
Pt este constructor de date

- combinație între sumă și produs

```
data List a = Nil
            | Cons a (List a)
```

Tipuri de date algebrice

Forma generală

$$\begin{aligned} data\ Typename\ = \quad & Cons_1\ t_{11} \ldots t_{1k_1} \\ & \mid Cons_2\ t_{21} \ldots t_{2k_2} \\ & \mid \ldots \\ & \mid Cons_n\ t_{n1} \ldots t_{nk_n} \end{aligned}$$

unde $k_1, \ldots, k_n \geq 0$

- Se pot folosi tipuri sumă și tipuri produs, tipuri parametrizate și definiții recursive.

Atenție!

Alternativele trebuie să conțină constructori.

```
data StrInt = String | Int -- este gresit

data StrInt = VS String | VI  Int -- este corect

[VI 1, VS "abc", VI 34, VI 0, VS "xyz"] :: [StrInt]
```

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Maybe a = Nothing | Just a

data Pair a b = Pair a b
  -- constructorul de tip si cel de date pot sa coincida

data Nat = Zero | Succ Nat

data Exp = Lit Int | Add Exp Exp | Mul Exp Exp

data List a = Nil | Cons a (List a)

data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
```

Exemplu - date personale. Utilizarea **type**

Cu **type** se pot redenumi tipuri deja existente.

```
type Name = String
type Age  = Integer

data Person = Person Name Age
```

Datele de descompun folosind proiecții:

```
name :: Person -> Name
name (Person name _ ) = name

age :: Person -> Age
age (Person _ years) = years
```

Date personale ca înregistrări

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     }
```

Proiecțiile sunt definite automat:

```
firstName :: Person -> String
lastName :: Person -> String
age :: Person -> Int
height :: Person -> Float
phoneNumber :: Person -> String
```

Date personale ca înregistrări

- Putem folosi atât forma algebrică cât și cea de înregistrare

```
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

gigel = Person { firstName = "Gheorghe"
               , lastName="Georgescu"
               , age = 30, height = 192.3
               , phoneNumber = "0798765432"
               }
```

- Putem folosi și pattern-matching
- Proiecțiile sunt definite automat; sintaxă specializată pentru actualizări

```
nextYear :: Person -> Person
nextYear person = person { age = age person + 1 }
```

Înregistrări cu funcții

```
newtype Calc = C {compute :: Int -> Int}
```

newtype se folosește când avem un singur cosntructor cu un singur argument.

Exemplu de utilizare

```
eval :: Calc -> Int -> Int
eval calc x = compute calc $ x
*Main> eval (C (\x -> x+1)) 3
4
```

Derivare automata pentru tipuri algebrice

Am definit tipuri de date noi:

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Eq, Ord, Show)

data Point a b = Pt a b
                 deriving (Eq, Ord, Show)
```

Cum putem să le facem instanțe ale claselor **Eq**, **Ord**, **Show**?

Putem să le facem explicit sau să folosim derivarea automată.

Atenție!
Derivarea automată poate fi folosită numai pentru unele clase predefinite.

Derivare automata pentru tipuri algebrice

```
data Point a b = Pt a b
                 deriving (Eq, Ord, Show)
```

Egalitatea, relația de ordine și modalitatea de afișare sunt definite implicit dacă este posibil:

```
*Main> Pt 2 3 < Pt 5 6
True

*Main> Pt 2 "b" < Pt 2 "a"
False

*Main Data.Char> Pt (+2) 3 < Pt (+5) 6

<interactive>:69:1: error:• No instance for (Ord (Integer -> Integer))arising from a
      use of ‘<’
```

Instanțiere explicită - exemplu

```
data Season = Spring | Summer | Autumn | Winter

eqSeason :: Season -> Season -> Bool
eqSeason Spring Spring = True
eqSeason Summer Summer = True
eqSeason Autumn Autumn = True
eqSeason Winter Winter = True
eqSeason _      _      = False


showSeason :: Season -> String
showSeason Spring = "Spring"
showSeason Summer = "Summer"
showSeason Autumn = "Autumn"
showSeason Winter = "Winter"
```

```
instance Eq Season where
    (==) = eqSeason
```

```
instance Show Season where
    show = showSeason
```

Instanțiere explicită - exemplu

```
data Season = Spring | Summer
            | Autumn | Winter

instance Enum Season where
  succ Spring = Summer
  succ Summer = Autumn
  succ Autumn = Winter
  succ Winter = Spring

  fromEnum Winter = 0
  fromEnum Spring = 1
  fromEnum Summer = 2
  fromEnum Fall   = 3

  toEnum 0 = Winter
  toEnum 1 = Spring
  toEnum 2 = Summer
  toEnum 3 = Fall
```

```
class Enum a where
  succ :: a -> a
  fromEnum :: a -> Int
  toEnum :: Int -> a
```

Exemplu: liste

Declarație ca tip de date algebric

```
data  List a  = Nil
              | Cons a (List a)

append :: List a -> List a -> List a
append Nil ys           = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Constructori simboluri

Declarație ca tip de date algebric cu simboluri

```
data  List a  = Nil
              | a ::: List a
          deriving (Show)
infixr 5 :::

(+++) :: List a -> List a -> List a
infixr 5 +++
Nil +++ ys        = ys
(x ::: xs) +++ ys = x ::: (xs +++ ys)
```

Comparați cu versiunea folosind notația predefinită

```
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Definirea egalității și a reprezentării

Eq și Show

```
eqList :: Eq a => List a -> List a -> Bool
eqList Nil Nil = True
eqList (x ::: xs) (y:::ys) = x == y && eqList xs ys
eqList _ _ = False

showList :: Show a => List a -> String
showList Nil = "Nil"
showList (x ::: xs) = show x ++ " ::: " ++ showList xs

instance (Eq a) => Eq (List a) where
  (==) = eqList

instance (Show a) => Show (List a) where
  show = showList
```

Cercuri și dreptunghiuri

```
type    Radius    =    Float
type    Width     =    Float
type    Height    =    Float

data    Shape     =    Circle Radius
                  |    Rectangle Width Height

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rectangle w h) = w * h
```

Definirea egalității și a reprezentării

Eq și Show

```
eqShape :: Shape -> Shape -> Bool
eqShape (Circle r) (Circle r') = (r == r')
eqShape (Rectangle w h) (Rectangle w' h') = (w == w') && (
    h == h')
eqShape _ _ = False

showShape :: Shape -> String
showShape (Circle r) = "Circle " ++ showF r
showShape (Rectangle w h) = "Rectangle " ++ showF w
    ++ " " ++ showF h

showF :: Float -> String
showF x | x >= 0 = show x
        | otherwise = "(" ++ show x ++ ")"
```

Teste și operatori de proiecție

```
isCircle :: Shape -> Bool
isCircle (Circle r) = True
isCircle _          = False

isRectangle :: Shape -> Bool
isRectangle (Rectangle w h) = True
isRectangle _               = False


area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rectangle w h) = w * h
```

Kinds (tipuri de tipuri)

Observăm că m în definiția de mai sus este un constructor de tip.

În Haskell, valorile sunt clasificate cu ajutorul *tipurilor*:

```
Prelude> :t "as"
"as" :: [Char]
```

Constructorii de tipuri sunt la rândul lor clasificați în *kind*-uri:

```
Prelude> :k Char
*     -- constructor de tip fara argumente
Prelude> :k []
[] :: * -> *   -- constructor de tip cu un argument
```

Cosntructorii de tip pot fi și ei grupați în clase.

Clasa de tipuri Functor - instanțe

```
class Functor m where
  fmap :: (a -> b) -> m a -> m b
```

- m este constructor de tip
- transformă fiecare element al colecției folosind f :: a−> b
- fără a afecta structura colecției

Observăm că o instanță a clasei **Functor** trebuie să fie un *constructor de tip* care are *kind-ul* $*->*$.

Instanță pentru liste

```
instance Functor [] where
  fmap = map
```

Clasa de tipuri Functor - instanțe

```
class Functor m where
  fmap :: (a -> b) -> m a -> m b
```

- Instanță pentru tipul opțiune

```
data Maybe a = Nothing
             | Just a

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

- Instanță pentru tipul arbore

```
data Arbore a = Nil
              | Nod a Arbore Arbore

instance Functor Arbore where
  fmap f Nil = Nil
  fmap f (Nod x l r) = Nod (f x) (fmap f l) (fmap f r)
```

Clasa de tipuri Functor - instanțe

```
class Functor m where
  fmap :: (a -> b) -> m a -> m b
```

- Un alt tip folosit pentru *opțiuni* este **Either**:

```
data Either e  a = Left e | Right a

mydiv :: Integer -> Integer ->  Either String Integer
mydiv  x y = if ( y==0)
                then Left "impartire la zero"
                else Right (x `div` y)
```

O instanță a clasei **Functor** trebuie să fie un *constructor de tip* care are *kind-ul* $*->*$, dar **Either** are un kind diferit.

```
Prelude> :k Either
Either :: * -> * -> *
```

Clasa de tipuri Functor - instanțe

```
class Functor m where
  fmap :: (a -> b) -> m a -> m b

data Either e  a = Left e | Right a
Either :: * -> * -> *  -- kind
```

Pentru a obține o instanță a clasei **Functor**, trebuie să fixăm un parametru în definiția tipului **Either**:

```
instance Functor (Either e) where
  fmap f (Left v) = Left v
  fmap f (Right x) = Right (f x)  -- f :: a -> b

-- fmap :: (a -> b) -> (Either e a) -> (Either e b)

*Main> fmap (\x -> x+1) (mydiv 3 4)
Right 1
*Main> fmap (\x -> x+1) (mydiv 3 0)
Left "impartire la zero"
```

Clasa de constructori - instanțe

Putem și noi să definim clase de constructori:

```
class (ToList c) where
  toList :: c a -> [a]   -- c este constructor de tip
  fromList :: c a -> [a]
```

Instanță pentru tipul **List**

```
data List a = Nil | Cons a (List a)

instance ToList List where
 toList Nil = []
 toList (Cons x xs ) = x   : (toList xs)
 fromList [] = Nil
 fromList (x:xs) = Cons x (fromList xs)

 *Main> toList (Cons 2 (Cons 1 Nil))
[2,1]
```

Monade

Monadele sunt functori

Formal, o *monadă* este o clasă de tipuri, care poate fi gândită ca un caz particular al clasei **Functor**. Mai exact, clasa monadelor poate fi definită prin următoarele operații:

```
fmap :: (a -> b) -> (m a -> m b)
join :: m (m a) -> m a
return :: a -> m a
```

```
*Main Control.Monad> join [[2]]
[2]
*Main Control.Monad> join (Just (Just 3))
Just 3
*Main Control.Monad> return 2 :: [Int]
[2]
*Main Control.Monad> return 2 :: Maybe Int
Just 2
```

Atenție! Nu aceasta este definiția monadelor în Haskell, ci una echivalentă.

Computații cu efect

In Haskell dorim:
- să păstrăm proprietatea de *referență transparențială*: o funcție întoarce aceeași valoare pentru aceleași intrări și nu are efecte laterale,
- să putem reprezenta computații cu efecte, de exemplu operații de intrare-ieșire, operații care depind de o stare, etc.

Să luăm ca exemplu instrucțiunea **getLine**. Observăm că **getLine** nu poate avea tipul **String** deoarece s-ar pierde transparența referențială. Soluția găsită în Haskell este următoarea
- valoarea lui **getLine** este o acțiune care, dacă va fi executată, va produce un **String**
- tipul lui **getLine** este **IO String**, unde **IO** este o monada (un constructor de tip din clasa **Monad**)

Despre rețete

inspirat de

https://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html

Să presupunem că avem două tipuri de date

```
p :: Prajitura  -- prajitura efectiva

rp :: Reteta Prajitura  -- descrierea a modului
                        -- in care facem prajitura
```

Dacă avem o Retata Prajitura, nu inseamna ca avem si Prajitura!
Înseamnă că, dacă executăm acțiunea (rețeta) vom obține o Prajitura.

În cazul monadei IO bucătarul, cel care este responsabil de execuția rețetelor, este mediul de execuție al limbajului Haskell.

Despre efecte și rețete

inspirat de

https://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html

Să presupunem că, având (gustând) o prajitură știm rețeta pentru o limonadă care se potrivește cu prajitura. Așadar avem o funcție

```
getLim :: Prajitura -> Reteta Limonada
```

Știind rp :: Reteta Prajitura vrem să obținem rl :: Reteta Limonada. Pentru aceasta ne trebuie o operație care să combine rp cu funcția de mai sus, adică să "execute" rețeta, să obțină prăjitura și să returneze rețeta de limonadă aleasă pe baza funcției:

```
>>= :: Reteta Prajitura -> (Prajitura -> Reteta Limonada)
                                    -> Reteta
                                       Limonada
```

Această operație se numește "bind" și este operația de baza a monadelor.

Monad - definiție alternativă

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

- În orice monadă putem defini

```
(>>) :: m a -> m b -> m b
-- nu se tine cont de rezultatele intermediare
x >> y = x >>= \_ -> y

join :: m(m a) -> m a          -- aplatizarea
join x = x >>= id

fmap :: (a -> b) -> (m a -> m b)
fmap f x = x >>= (return . f) -- monadele sunt functori
```

Despre Applicative vom discuta mai târziu!

Notația „do” pentru monade

- e >>= \x ->e1 devine x <- e ; e1
- e >> e1 devine e ; e1

De exemplu

```
e1    >>= \x1 -> e2    >>= \x2 -> e3
```

este echivalent cu

```
do {x1 <- e1;
    x2 <- e2;
    e3}
```

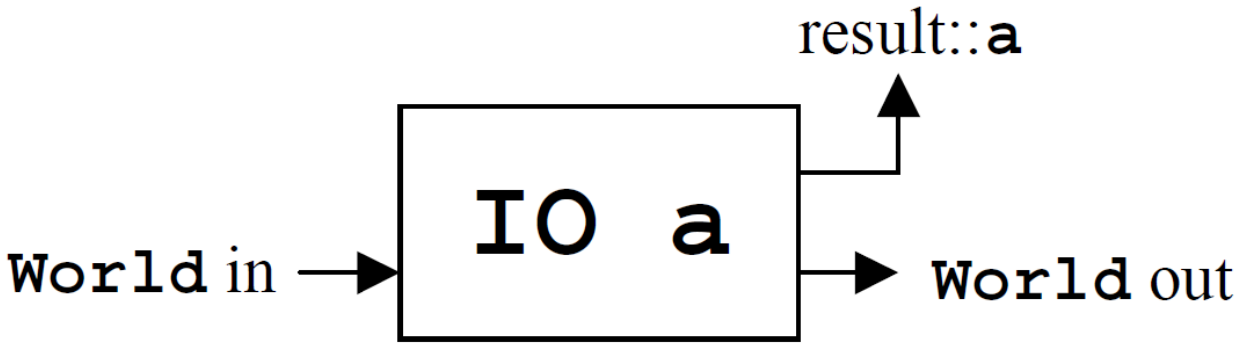
Obervăm că x <- e1 nu are sens separat, trebuie sa fie urmata de o continuare; ultima expresie dintr-un bloc **do** trebuie sa intoarca un tip monadic (m a).

Exemple de efecte laterale

I/O	Monada `IO`
Logging	Monada `Writer`
Stare	Monada `State`
Excepții	Monada `Either`
Parțialitate	Monada `Maybe`
Nedeterminism	Monada `[]` (listă)
Memorie read-only	Monada `Reader`

Monada **IO**

```
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Comenzi cu valori

- **IO** () corespunde comenzilor care nu produc rezultate

```
putChar :: Char -> IO ()
putStr  :: String -> IO ()
putStrLn :: String -> IO ()
```

- În general, **IO** a corespunde comenzilor care produc rezultate de tip a.

```
getChar :: IO Char
getLine  :: IO String
```

Compilarea programelor

main

Orice comandă **IO a** poate fi executată în interpretor, dar

Programele Haskell pot fi compilate

Fișierul scrie.hs:

```
main :: IO ()
main = putStrLn "?!"
```

```
08-io$ ghc scrie.hs
[1 of 1] Compiling Main  (scrie.hs, scrie.o)
Linking scrie.exe ...
08-io$ ./scrie
?!
```

Funcția executată este **main**

Monada **IO** -Exemple

```
putStr :: String -> IO ()
putStr []     = done
putStr (x:xs) = putChar x >> putStr xs

putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'

getLine :: IO String
getLine = do {
            x <- getChar;
            if x == '\n' then
              return []
            else do {
              xs <- getLine;
              return (x:xs)
            }
          }
```

Monada Writer (variantă simplificată)

```
newtype Writer log a = Writer { runWriter :: (a, log) }
-- a este parametru de tip

tell :: log -> Writer log ()
tell msg = Writer ((), msg)


instance Monad (Writer String) where
  return a = Writer (a, "")     -- a este variabila
  ma >>= k = let (a, log1) = runWriter ma
                 (b, log2) = runWriter (k a)
             in Writer (b, log1 ++ log2)
```

Observație

În definițiile de mai sus am folosit a pentru a desemna un tip, dar și o variabilă de acel tip, contextul în care apare clarifică modul de utilizare. Convenții similare: b este o variabilă de tip b, log1 si log2 sunt variabile de tip **log**, iar ma este o variabilă de tip monadic (m a).

Monada Writer -Exemplu logging

```
newtype Writer log a = Writer { runWriter :: (a, log) }

tell :: log -> Writer log ()
tell msg = Writer ((), msg)


logIncrement :: Int -> Writer String Int
logIncrement x = do
  tell ("increment: " ++ show x ++ "\n")
  return (x + 1)

logIncrement2 :: Int -> Writer String Int
logIncrement2 x = do
  y <- logIncrement x
  logIncrement y

Main> runWriter (logIncrement2 13)
(15,"increment: 13\nincrement: 14\n")
```

Monada State

```
newtype State state a = State{runState :: state ->(a,state)}
```

O variabilă ma :: State state a va avea una din formele
ma = State f *sau* ma = State {runState =f} cu f :: state -> (a, state)

```
instance Monad (State state) where
  return a = State (\s -> (a, s))
-- return a = State f where f = \s -> (a,s)

  ma >>= k = State g
    where g state = let (a, aState) = runState ma state
                    in runState (k a) aState

-- ma :: State state a , runState ma :: state -> (a, state)
-- k :: a -> State state b,  runState (k a) :: state -> (b,
    state)
-- ma >>= k :: State state b
--  g :: state -> (b, state)
```

Monada State

```
newtype State state a = State {runState :: state ->(a, state
    )}

instance Monad (State state) where
  return a = State (\ s -> (a, s))
  ma >>= k = State g
    where g state = let (a, aState) = runState ma state
                    in runState (k a) aState
```

Funcții ajutătoare:

```
get :: State state state
get = State (\s -> (s, s))

modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))
```

Monada State - exemplu random

```
newtype State state a = State{runState :: state ->(a,state)}
get :: State state state
get = State (\s -> (s,s))
modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))

cMULTIPLIER, cINCREMENT :: Word32
cMULTIPLIER = 1664525 ; cINCREMENT = 1013904223
rnd, rnd2 :: State Word32 Word32

rnd = do modify (\seed -> cMULTIPLIER * seed + cINCREMENT)
         get
rnd2 = do r1 <- rnd
          r2 <- rnd
          return (r1 + r2)

Main> runState rnd2 0
(2210339985,1196435762)
```

Monada **Either** (a excepțiilor)

```
data Either err a = Left err | Right b

instance Monad (Either err) where
  return = Right
  Right a >>= k  = k a
  err     >>= _  = err

radical :: Float -> Either String Float
radical x | x >= 0 = return (sqrt x)
          | x < 0  = Left "radical: argument negativ"

solEq2 :: Float -> Float -> Float -> Either String Float
solEq2 0 0 0 = return 0                    -- a * x^2 + b * x +
    c = 0
solEq2 0 0 c = Left "Nu are solutii"
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do let delta = b * b - 4 * a * c
                  rDelta <- radical delta
                  return (negate b + rDelta) / (2 * a)
```

Monada **Maybe** (a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just a  >>= k    = k a
  Nothing >>= _    = Nothing

radical :: Float -> Maybe Float
radical x | x >= 0 = return (sqrt x)
          | x < 0  = Nothing

solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0                    -- a * x^2 + b * x +
    c = 0
solEq2 0 0 c = Nothing
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do let delta = b * b - 4 * a * c
                  rDelta <- radical delta
                  return (negate b + rDelta) / (2 * a)
```

Monada listelor (a funcțiilor nedeterministe)

```
instance Monad [] where
  return a = [a]
  ma >>= k = [b | a <- ma, b <- k a]
```

Rezultatul funcției e lista tuturor valorilor posibile.

```
radical :: Float -> [Float]
radical x | x >= 0 = [negate (sqrt x), sqrt x]
          | x < 0  = []

solEq2 :: Float -> Float -> Float -> [Float]
solEq2 0 0 c = []                          -- a * x^2 + b * x + c
    = 0
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do let delta = b * b - 4 * a * c
                  rDelta <- radical delta
                  return (negate b + rDelta) / (2 * a)
```

Monada Reader (stare nemodificabilă)

```
newtype Reader env a = Reader { runReader :: env -> a }

ask :: Reader env env
ask = Reader id

instance Monad (Reader env) where
  return = Reader const  -- return x = Reader (\_ -> x)
  ma >>= k = Reader f
    where f env = let a = runReader ma env
                  in  runReader (k a) env
```

Monada Reader - exemplu: mediu de evaluare

```
newtype Reader env a = Reader { runReader :: env -> a }
ask :: Reader env env
ask = Reader id

data Prop ::= Var String | Prop :&: Prop
type Env = [(String, Bool)]

var :: String -> Reader Env Bool
var x = do
  env <- ask
  fromMaybe False (lookup x env)

eval :: Prop -> Reader Env Bool
eval (Var x) = var x
eval (p1 :&: p2) = do
  b1 <- eval p1
  b2 <- eval p2
  return (b1 && b2)
```

Functor și Applicative pot fi definiți cu **return** și >>=

```
instance Monad M where
  return a = ...
  ma >>= k = ...
```

Datorită constrângerilor de tip, o instanță a clasei **Monad** trebuie să fie și instantță a claselor Applicative și **Functor**, dar acestea pot fi scrise folosind operațiile monadice astfel:

```
instance Applicative M where
  pure = return
  mf <*> ma = do
                f <- mf
                a <- ma
                return (f a)
-- mf <*> ma = mf >>= (\f -> ma >>= (\a -> return (f a)))

instance Functor F where          -- ma >>= \a -> return (f a)
  fmap f ma = pure f <*> ma       -- ma >>= (return . f)
```

Pe săptămâna viitoare!