

## PROGRAMARE FUNCTIONALA

### COMENTARII

-- comentariu pe o linie  
{- comentariu pe mai multe linii -}

Variabilele sunt imutabile! Operatorul = nu este operator de atribuire,  $x = 1$  reprezinta o legatura. Din momentul in care o variabila este legata de o valoare, acea valoare nu mai poate fi schimbata.

Expresia **LET .. IN ..** este o expresie care creeaza scop local.  
Acesta vede cea mai apropiata definitie si o atribuie.

$x = 1$ $z = \text{let } x = 1 \text{ in } x$	$x = \text{let}$ $z = 5$ $g\ u = z + u$ in let $z = 7$ in $g\ 0 + z$	$x = \text{let } z = 5; g\ u = z + u \text{ in let } z = 7 \text{ in } g\ 0$	$x = [\text{let } y = 8 \text{ in } y, 9]$
$z = 3$ $x = 1$	$x = 12$	$x = 5$	$x = [8, 9]$

Clauza **WHERE** creeaza scop local, fiind disponibila doar la nivel de definitie.

$f\ x = g\ x + g\ x + z$ where $g\ x = 2 * x$ $z = x - 1$	$x = [y \text{ where } y = 8, 9]$
Codul este corect!	Error: parse error ...

Variabile pot fi legate si prin pattern matching la definirea unei functii sau expresii case.

$h\ x \quad   \ x == 0 = 0$ $  \ x == 1 = y + 1$ $  \ x == 2 = y * y$ $  \ \text{otherwise} = y$ where $y = x * x$	$f\ x = \text{case } x \text{ of}$ $0 \rightarrow 0$ $1 \rightarrow y + 1$ $2 \rightarrow y * y$ $\_ \rightarrow y$ where $y = x * x$
--	--

### TIPURI DE DATE:

- Tipuri de baza: Int, Integer, Float, Double, Bool, Char, String
- Tipuri compuse: tupluri si liste

- c. Tipuri noi definite de utilizator
- ```
data RGB = Red | Green | Blue
data Point a = Pt a a
-- tip parametrizat, a este variabila de tip
```

Pentru a afla tipul:

|                                                            |                                                                   |
|------------------------------------------------------------|-------------------------------------------------------------------|
| Prelude> :t ( 'a', True)<br>( 'a', True ) :: (Char, Bool ) | Prelude> :t [ "ana", "ion" ]<br>[ "ana", " ion" ] :: [ [ Char ] ] |
| Un tuplu poate avea elemente de tipuri diferite.           | O lista are elemente de acelasi tip.                              |

Prelude> :t 1

1 :: Num a => a

Semnificatia este urmatoarea: Num este o clasa de tipuri, a este un parametru de tip, iar 1 este o valoare de tipul a din clasa Num.

Prelude> :t [1, 2, 3]

[1, 2, 3] :: Num t => [ t ]

## LISTE

Definitie recursiva. O **lista** este:

- vida, notata [], sau
- compusa, notata x:xs, dintr-un element x capul listei (head) si o lista xs coada listei (tail)

Orice lista poate fi scrisa folosind doar constructorul (:) si lista vida [].

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

Constructorul pentru liste este (:), iar constructorul pentru tupluri este (,).

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

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

Prelude> x

1

Prelude> y

[2, 3]

A avut loc deconstructia valorii [1, 2, 3] in 1 : [2, 3] si legarea lui x la 1 si a lui y la [2, 3].

X:XS se potriveste doar cu listele nevide!

Definitii folosind sabloane:

reverse [] = []

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

Sabloanele sunt definite folosind constructori. De exemplu, operatia de concatenare pe liste este:

(++) :: [a] -> [a] -> [a], dar [x] ++ [1] = [2, 1] nu va avea ca efect legarea lui x la 2.

```
Prelude> [x] ++ [1] = [2, 1]
```

```
Prelude> x
```

---

Eroare

Sabloanele sunt liniare, adica o variabila apare cel mult odata. Sabloane în care o variabila apare de mai multe ori provoaca mesaje de eroare. De exemplu:

```
x : x : [1] = [2, 2, 1]
```

```
ttail (x : x : t) = t
```

```
foo x x = x^2
```

---

error: Conflicting definitions for x

O solutie este folosirea garzilor:

|                                                     |                                               |
|-----------------------------------------------------|-----------------------------------------------|
| ttail (x : y : t)   (x==y) = t<br>  otherwise = ... | foo x y   (x == y) = x^2<br>  otherwise = ... |
|-----------------------------------------------------|-----------------------------------------------|

### INTERVALE, PROGRESII FINITE, PROGRESII INFINITE, LISTE INFINITE

|                                                        |                                                             |                                                                  |
|--------------------------------------------------------|-------------------------------------------------------------|------------------------------------------------------------------|
| interval = ['c'..'e']<br>['c', 'd', 'e']               | progresie_finita = [20, 17..1]<br>[20, 17, 14, 11, 8, 5, 2] | progresie_finita = [2.0, 2.5,..4.0]<br>[2.0, 2.5, 3.0, 3.5, 4.0] |
| lista_infinita = [0..]<br>[0, 1, 2, 3, 4, ... infinit] | progresie_infinita = [0, 2 ..]<br>[0, 2, 4, 6, ... infinit] |                                                                  |

```
Prelude> natural = [0..]
```

```
Prelude> take 5 natural
```

---

```
[0, 1, 2, 3, 4]
```

Laziness – argumentele sunt evaluate doar cand este necesar si doar cat este necesar.

```
Prelude> evenNat = [0, 2 ..]
```

```
Prelude> take 7 evenNat
```

---

```
[0, 2, 4, 6, 8, 10, 12]
```

O lista este indexata de la 0. Pentru a accesa un element de pe o pozitie data folosim (!!).

```
Prelude> [1, 2, 3] !! 2
```

---

```
3
```

```
Prelude> "abcd " !! 0
```

---

```
'a'
```

Pentru a adauga un element la o lista folosim (++).

```
Prelude> [1, 2] ++ [3]
```

---

```
[1, 2, 3]
```

Definitia prin selectie a unei liste:

$[E(x) \mid x \leftarrow [x_1, \dots, x_n], P(x)]$

```
Prelude> xs = [0..10]
```

```
Prelude> [x | x <- xs, even x]
```

---

```
[0, 2, 4, 6, 8, 10]
```

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

Putem folosi si let pentru declarati 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)]
```

## ZIP

Functia zip se utilizeaza pe doua liste si face o lista de tupluri dupa cum urmeaza:

- (primul caracter  $L_1$ , primul caracter  $L_2$ ), (al doilea caracter  $L_1$ , al doilea caracter  $L_2$ ) etc.

Se va opri la lungimea listei mai mici.

```
Prelude> ys = ['A'..'E']
Prelude> zip [1..] ys
[(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'), (5, 'E')]
```

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

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

Atentie! Se observa o diferenta intre cele doua comenzi.

```
Prelude> zip [1..3] ['A'..'D']
[(1, 'A'), (2, 'B'), (3, 'C')]
```

```
Prelude> [(x, y) | x <- [1..3], y <- ['A'..'D']]
[(1, 'A'), (1, 'B'), (1, 'C'), (1, 'D'), (2, 'A'), (2, 'B'), (2, 'C'), (2, 'D'), (3, 'A'), (3, 'B'), (3, 'C'), (3, 'D')]
```

## FUNCTII

- Prototipul functiei (nume + signatura)
- Definitia functiei (nume + parametru formal + corp)
- Aplicarea functiei (nume + parametru actual)

|                                                  |                                                         |                                                                    |
|--------------------------------------------------|---------------------------------------------------------|--------------------------------------------------------------------|
| double :: Integer -> Integer<br>double a = a + a | add :: Integer -> Integer -> Integer<br>add a b = a + b | dist :: (Integer, Integer) -> Integer<br>dist (a, b) = abs (a - b) |
| double 5                                         | add 3 7                                                 | dist(5, 7)                                                         |

Putem defini o functie folosind if, ecuatii, cazuri:

fact :: Integer -> Integer

| IF                                             | ECUATII                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | CAZURI                                              |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| fact n = if n == 0 then 1<br>else n*fact (n-1) | fact 0 = 1<br>fact n = n*fact (n-1)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | fact n<br>  n == 0 = 1<br>  otherwise = n*fact(n-1) |
|                                                | <p>Variabilele si valorile din partea stanga a semnului = sunt sabloane (0 si n). Cand functia este apelata se incearca potrivirea parametrilor actuali cu sabloanele, ecuatiile fiind incercate in ordinea scrierii (0 se potriveste cu el insusi, iar n cu orice valoare de tip Integer).</p> <p>Atentie! Daca schimbam ordinea ecuatiilor din definitia factorialului, functia nu isi va incheia executia! Deoarece n e un pattern care se potriveste cu orice valoare, inclusive cu 0, orice apel al functiei va merge pe prima ecuatie.</p> |                                                     |

Sabloane:

selectie :: Integer -> String -> String

|                                                                                                     |                                                                                          |
|-----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| selectie x s = case (x, s) of<br>(0, _) -> s<br>(1, z:zs) -> zs<br>(1, [ ]) -> [ ]<br>_ -> (s ++ s) | selectie 0 s = s<br>selectie 1 (_:s) = s<br>selectie 1 " " = " "<br>selectie _ s = s + s |
| Sablonul _ se numeste wild-card pattern si se potriveste cu orice valoare.                          |                                                                                          |

Fie foo o functie.

|                                                                                                                                                                      |                                                                                                                                                                                                                                               |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| foo :: a -> b -> [ a ] -> [ b ] <ul style="list-style-type: none"> <li>are trei argumente, de tipuri a, b si [a]</li> <li>întoarce un rezultat de tip [b]</li> </ul> | foo :: ( a -> b ) -> [ a ] -> [ b ] <ul style="list-style-type: none"> <li>are doua argumente, de tipuri (a -&gt; b) si [a], adica o functie de la a la b si o lista de elemente de tip a</li> <li>întoarce un rezultat de tip [b]</li> </ul> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## OPERATORI

Operatorii in Haskell au doua argumente.

Acestia pot fi apelati folosind notatia infix, folosind paranteze.

Prelude> (+) 2 3

Operatorii care sunt definiti in forma prefix sunt apelati in forma infix folosind backtick (` `).

|                  |                    |
|------------------|--------------------|
| Prelude> mod 5 2 | Prelude> 5 `mod` 2 |
| 1                | 1                  |

elem :: a -> [a] -> Bool

Prelude> 1 `elem` [1, 2, 3]

True

Operatorii pot fi definiti folosind numai simboluri.

1. Operatori predefiniti

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

2. Operatori definiti de utilizator

(&&) :: Bool -> Bool -> Bool

True && b = b

False && \_ = False

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          |                  |   |       |       |       |        |                 |    |   |           |   |    | \$                | \$! | `seq` |

Exemple:

- Operatorul (-) este asociativ la stanga:  
5 - 2 - 1 == (5 - 2) - 1
- Operatorul (:) este asociativ la dreapta:  
5 : 2 : [] == 5 : (2 : [])
- Operatorul (++) este asociativ la dreapta:  
(++) :: [a] -> [a] -> [a]  
[] ++ ys = ys  
(x : xs) ++ ys = x : (xs ++ ys)  
L<sub>1</sub> ++ L<sub>2</sub> ++ L<sub>3</sub> ++ L<sub>4</sub> ++ L<sub>5</sub> == L<sub>1</sub> ++ (L<sub>2</sub> ++ (L<sub>3</sub> ++ (L<sub>4</sub> ++ L<sub>5</sub>)))

## SECTIUNI

Sectiunile operatorului binar (op) sunt (op e) si (e op).

Sectiunile lui (++) sunt (++ e) si (e ++).

|                           |                                       |
|---------------------------|---------------------------------------|
| Prelude> :t (++)          | Prelude> :t (++ " world!")            |
| (++) :: [a] -> [a] -> [a] | (++ " world ! " ) :: [Char] -> [Char] |

|                                 |                               |
|---------------------------------|-------------------------------|
| Prelude> (++ " world!") "Hello" | Prelude> ++ " world!" "Hello" |
| "Hello world!"                  | error                         |

Sectiunile lui (<->) sunt (<-> e) si (e <->).

Prelude> x <-> y = x - y + 1

Prelude> :t (<-> 3)

(<-> 3) :: Num a => a -> a

Prelude> (<-> 3) 4

2

Sectiunile sunt afectate de asociativitatea si precedenta operatorilor.

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

## FUNCTII DE NIVEL INALT

Funcitiile pot fi folosite ca argumente pentru alte functii. Funcitiile anonime se numesc lambda expresii.

Forma generala a unei expresii:  $\lambda x_1 x_2 \dots x_n$

Prelude> inc = \x -> x+1

Prelude> add = \x y -> x+y

Prelude> aplic = \f x -> f x

|                        |                                       |
|------------------------|---------------------------------------|
| Prelude> (\x -> x+1) 3 | Prelude> map (\x -> x+1) [1, 2, 3, 4] |
| 4                      | [2, 3, 4, 5]                          |

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

| Lambda Expresii        | Sabloane           | Flip ca valoare de tip functie |
|------------------------|--------------------|--------------------------------|
| flip f = \x y -> f y x | flip f x y = f y x | flip = \f x y -> f y x         |

Compunerea functiilor se realizeaza cu operatorul (.).

Date fiind f : A -> B si g : B -> C, compunerea lor, notata g o f : A -> C este data de (g o f)(x) = g(f(x)).

(.) :: (b -> c) -> (a -> b) -> (a -> c)

(g . f) x = g (f x)

|                           |                                |
|---------------------------|--------------------------------|
| Prelude> :t reverse       | Prelude> :t take 5 . reverse   |
| reverse :: [a] -> [a]     | take 5 . reverse :: [a] -> [a] |
| Prelude> :t take          |                                |
| take :: Int -> [a] -> [a] |                                |

Prelude> (take 5 . reverse) [1..10]

[10, 9, 8, 7, 6]

Operatorul (\$) are precedenta 0 si este asociativ la dreapta.

(\$) :: (a -> b) -> a -> b

f \$ x = f x

|                                            |                                             |                                               |
|--------------------------------------------|---------------------------------------------|-----------------------------------------------|
| Prelude> (head . reverse . take 5) [1..10] | Prelude> head . reverse . take 5 \$ [1..10] | Prelude> head \$ reverse \$ take 5 \$ [1..10] |
| 5                                          | 5                                           | 5                                             |

## MAP

MAP se foloseste pentru transformarea fiecarui element dintr-o lista.

Date fiind o functie de transformare si o lista, aplicati functia fiecarui element al unei liste date.

| Solutie descriptiva                                         | Solutie recursiva                                                                 |
|-------------------------------------------------------------|-----------------------------------------------------------------------------------|
| map :: (a -> b) -> [a] -> [b]<br>map f xs = [f x   x <- xs] | map :: (a -> b) -> [a] -> [b]<br>map f [] = []<br>map f (x : xs) = f x : map f xs |

Definiti o functie care pentru o lista de numere întregi data ridica la patrat fiecare element din lista.

| Solutie descriptiva                                       | Solutie recursiva                                                                   | Solutie MAP                                                               |
|-----------------------------------------------------------|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| squares :: [Int] -> [Int]<br>squares xs = [x*x   x <- xs] | squares :: [Int] -> [Int]<br>squares [] = []<br>squares (x : xs) = x*x : squares xs | squares :: [Int] -> [Int]<br>squares xs = map sqr xs<br>where sqr x = x*x |

Prelude> squares [1, -2, 3]

[1, 4, 9]

Transformati un sir de caractere în lista codurilor ASCII ale caracterelor.

| Solutie descriptiva                                    | Solutie recursiva                                                          | Solutie MAP                                     |
|--------------------------------------------------------|----------------------------------------------------------------------------|-------------------------------------------------|
| ords :: [Char] -> [Int]<br>ords xs = [ord x   x <- xs] | ords :: [Char] -> [Int]<br>ords [] = []<br>ords (x : xs) = ord x : ords xs | ords :: [Char] -> [Int]<br>ords xs = map ord xs |

Prelude> ords "a2c3"

[97, 50, 99, 51]

map :: (a->b) -> [a] -> [b]

map f l = [f x | x <- l]



```
Prelude> map ($) [(4 +), (10 *), (^2), sqrt]
```

```
[7.0, 30.0, 9.0, 1.7320508075688772]
```

-- in acest caz, primul argument este o sectiune a operatorului (\$), iar al doilea o lista de functii

-- map(\$x) [f<sub>1</sub>, f<sub>2</sub>, ... f<sub>n</sub>] == [f<sub>1</sub>x, f<sub>2</sub>x, ... f<sub>n</sub>x]

## FILTER

Filter este folosit pentru selectarea elementelor dintr-o lista.

Date fiind un predicat (functie booleana) si o lista, selectati elementele din lista care satisfac predicatul.

| Solutie descriptiva                                                                                           | Solutie recursiva                                                                                                                                                                                              |
|---------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>filter :: (a -&gt; Bool) -&gt; [a] -&gt; [a]</code><br><code>filter p xs = [x   x &lt;- xs, p x]</code> | <code>filter :: (a -&gt; Bool) -&gt; [a] -&gt; [a]</code><br><code>filter p [] = []</code><br><code>filter p (x : xs)</code><br>  <code>p x = x : filter p xs</code><br>  <code>otherwise = filter p xs</code> |

Definiti o functie care selecteaza cifrele dintr-un sir de caractere.

| Solutie descriptiva                                                                                | Solutie recursiva                                                                                                                                                                             | Solutie FILTER                                                                           |
|----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| <code>digits :: [Char] -&gt; [Char]</code><br><code>digits xs = [x   x &lt;- xs, isDigit x]</code> | <code>digits :: [Char] -&gt; [Char]</code><br><code>digits [] = []</code><br><code>digits (x : xs)</code><br>  <code>isDigit x = x : digits xs</code><br>  <code>otherwise = digits xs</code> | <code>digits :: [Char] -&gt; [Char]</code><br><code>digits xs = filter isDigit xs</code> |

```
Prelude> digits "a2c3"
```

```
"23"
```

Definiti o functie care selecteaza elementele pozitive dintr-o lista.

| Solutie descriptiva                                                                                   | Solutie recursiva                                                                                                                                                                                         | Solutie FILTER                                                                                                                  |
|-------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <code>positives :: [Int] -&gt; [Int]</code><br><code>positives xs = [x   x &lt;- xs, x &gt; 0]</code> | <code>positives :: [Int] -&gt; [Int]</code><br><code>positives [] = []</code><br><code>positives (x : xs)</code><br>  <code>x &gt; 0 = x : positives xs</code><br>  <code>otherwise = positives xs</code> | <code>positives :: [Int] -&gt; [Int]</code><br><code>positives xs = filter pos xs</code><br>where <code>pos x = x &gt; 0</code> |

```
Prelude> positives [1, -2, 3]
```

```
[1, 3]
```

## CURRYING

Currying este procedeul prin care o functie cu mai multe argumente este transformata intr-o functie care are un singur argument si intoarce o alta functie. In Haskell toate functiile sunt in forma curry, deci au un singur argument.

Operatorul (->) pe tipuri este asociativ la dreapta: `a1 -> a2 -> ... -> an == a1 -> (a2 -> ... (an-1 -> an)...).`

Aplicarea functiilor este asociativa la stanga: `f x1 ... xn == (...((f x1)x2)...xn).`

|                                      |                                        |
|--------------------------------------|----------------------------------------|
| Prelude> :t curry                    | Prelude> :t uncurry                    |
| curry :: ((a,b) -> c) -> a -> b -> c | curry :: ((a -> b -> c) -> (a, b) -> c |

Exemplu:

|                                                     |                             |
|-----------------------------------------------------|-----------------------------|
| f :: (Int, String) -> String<br>f (n, s) = take n s | Prelude> f(1, "abc")<br>"a" |
| Prelude> let cf = curry f<br>Prelude> :t cf         | Prelude> cf 1 "abc"<br>"a"  |
| cf :: Int -> String -> String                       |                             |

Fie  $f : A \times B \rightarrow C$  o functie. In mod uzual scriem  $f(x, y) = z$  unde  $x$  apartine  $A$ ,  $y$  apartine  $B$  si  $z$  apartine  $C$ . Pentru un  $x$  din  $A$  (arbitrar, fixat) definim  $f_x : B \rightarrow C$ ,  $f_x(y) = z$  daca si numai daca  $f(x, y) = z$ . Functia  $f_x$  se obtine prin aplicarea partiala a functiei  $f$ .

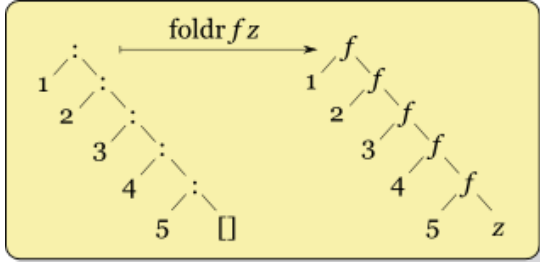
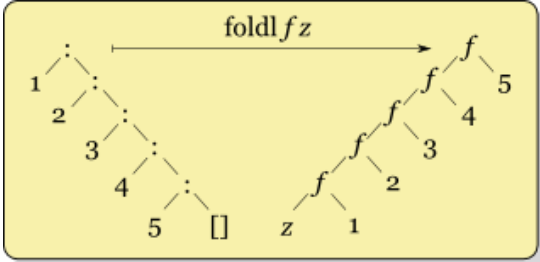
Daca notam  $B \rightarrow C = \{h : B \rightarrow C \mid h \text{ functie}\}$ , observam ca  $f_x$  apartine  $B \rightarrow C$  pentru orice  $x$  din  $A$ .

Asociem lui  $f$  functia  $cf : A \rightarrow (B \rightarrow C)$ ,  $cf(x) = f_x$ .

Vom spune ca functia  $cf$  este forma curry a functiei  $f$ .

### AGREGAREA ELEMENTELOR DINTR-O LISTA

Date fiind o functie de actualizare a valorii calculate cu un element curent, o valoare initiala si o lista, calculate valoarea obtinuta prin aplicarea repetata a functiei de actualizare fiecarui element din lista.

| FOLDR                                                                                                                                                                                       | FOLDL                                                                                                                                                                                          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| foldr :: (a -> b -> b) -> b -> [a] -> b<br>foldr f i [] = i<br>foldr f i (x:xs) = f x (foldr f i xs)                                                                                        | foldl :: (b -> a -> b) -> b -> [a] -> b<br>foldl h i [] = i<br>foldl h i (x:xs) = foldl h (h i x) xs                                                                                           |
|                                                                                                          |                                                                                                            |
| foldr op z [a <sub>1</sub> , a <sub>2</sub> , a <sub>3</sub> , ..., a <sub>n</sub> ] =<br>a <sub>1</sub> `op` (a <sub>2</sub> `op` (a <sub>3</sub> `op` (... (a <sub>n</sub> `op` z) ...))) | foldl op z [a <sub>1</sub> , a <sub>2</sub> , a <sub>3</sub> , ..., a <sub>n</sub> ] =<br>(... (((z `op` a <sub>1</sub> ) `op` a <sub>2</sub> ) `op` a <sub>3</sub> ) ...) `op` a <sub>n</sub> |
| Elemente procesate de la dreapta la stanga.<br>sum [x <sub>1</sub> , ... x <sub>n</sub> ] = (x <sub>1</sub> + (x <sub>2</sub> + ... (x <sub>n</sub> + 0) ...))                              | Elemente procesate de la stanga la dreapta.<br>sum [x <sub>1</sub> , ... x <sub>n</sub> ] = (... (0 + x <sub>1</sub> ) + x <sub>2</sub> ) + ... x <sub>n</sub> )                               |

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

| Solutie recursiva                                                                                    | Solutie recursive cu operator infix                                                                        |
|------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| foldr :: (a -> b -> b) -> b -> [a] -> b<br>foldr f i [] = i<br>foldr f i (x:xs) = f x (foldr f i xs) | foldr :: (a -> b -> b) -> b -> [a] -> b<br>foldr op i [] = i<br>foldr op i (x:xs) = x `op` (foldr op i xs) |

In definitia lui foldr, b poate fi tipul unei functii.

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

```
Prelude> compose [(+1), (^2)] 3
```

```
10
```

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

Definiti o functie care data fiind o lista cu numere intregi calculeaza suma elementelor dintr-o lista.

| Solutie recursiva 1                                                                                    | Solutie FOLDR                                  |
|--------------------------------------------------------------------------------------------------------|------------------------------------------------|
| sum :: [Int] -> Int<br>sum [] = 0<br>sum (x:xs) = x + sum xs                                           | sum :: [Int] -> Int<br>sum xs = foldr (+) 0 xs |
|                                                                                                        | sum :: [Int] -> Int<br>sum = foldr (+) 0       |
| Solutie recursiva 2                                                                                    | Solutie FOLDL                                  |
| sum :: [Int] -> Int<br>sum xs = suml xs 0<br>where<br>suml [] n = n<br>suml (x : xs) n = suml xs (n+x) | sum :: [Int] -> Int<br>sum xs = foldl (+) 0 xs |
|                                                                                                        | sum :: [Int] -> Int<br>sum = foldl (+) 0       |

```
Prelude> sum [1, 2, 3, 4]
```

```
10
```

Explicatii:

- Cu foldr (+) 0 [1, 2, 3] == 1 + (2 + (3 + 0)), elementele sunt procesate de la dreapta la stanga. Acelasi lucru se intampla si in solutia recursiva 1.
- Cu foldl (+) 0 [1, 2, 3] == 1 + (2 + (3 + 0)), elementele sunt procesate de la stanga la dreapta. Acelasi lucru se intampla si in solutia recursiva 2.

Definiti o functie care data fiind o lista cu numere intregi calculeaza produsul elementelor dintr-o lista.

| Solutie recursiva                                                            | Solutie FOLDR                                          |
|------------------------------------------------------------------------------|--------------------------------------------------------|
| product :: [Int] -> Int<br>product [] = 1<br>product (x:xs) = x * product xs | product :: [Int] -> Int<br>product xs = foldr (*) 1 xs |

```
Prelude> product [1, 2, 3, 4]
```

```
24
```

Explicatie: foldr (\*) 1 [1, 2, 3] == 1 \* (2 \* (3 \* 1))

Definiti o functie care concateneaza o lista de liste.

| Solutie recursiva                                                              | Solutie FOLDR                                          |
|--------------------------------------------------------------------------------|--------------------------------------------------------|
| concat :: [[a]] -> [a]<br>concat [] = []<br>concat (xs:xss) = xs ++ concat xss | concat :: [Int] -> Int<br>concat xs = foldr (++) [] xs |

```
Prelude> concat [[1, 2, 3], [4, 5]]
```

```
[1, 2, 3, 4, 5]
```

```
Prelude> concat ["con", "ca", "te", "na", "re"]
```

```
"concatenare"
```

```
Explicatie: foldr (++) [] ["Ana", "are", "mere."] == "Ana" ++ ("are" ++ ("mere." ++ []))
```

Calculati suma patratelor numerelor positive.

| Solutii fara FOLDR                                                  |                                                                         |                                                                                                         |
|---------------------------------------------------------------------|-------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| <pre>f :: [Int] -&gt; Int f xs = sum (squares (positives xs))</pre> | <pre>f :: [Int] -&gt; Int f xs = sum [x*x   x &lt;- xs, x &gt; 0]</pre> | <pre>f :: [Int] -&gt; Int f [] = 0 f (x:xs)   x &gt; 0 = (x*x) + f xs            otherwise = f xs</pre> |

Calculati suma patratelor numerelor positive.

| Solutii cu FOLDR                                                                                                            |                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <pre>f :: [Int] -&gt; Int f xs = foldr (+) 0 (map sqr (filter pos xs))   where sqr x = x * x         pos x = x &gt; 0</pre> | <pre>f :: [Int] -&gt; Int f xs = foldr (+) 0       (map (\x -&gt; x*x)        (filter (\x -&gt; x &gt; 0) xs))</pre> |
| <pre>f :: [Int] -&gt; Int f xs = foldr (+) 0 (map (^2) (filter (&gt;0) xs))</pre>                                           | <pre>f :: [Int] -&gt; Int f = foldr (+) 0 . map (^2) . filter (&gt;0)</pre>                                          |

Definiti o functie care data fiind o lista de elemente, calculeaza lista in care elementele sunt scrise in ordine inversa.

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

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

```
-- flip (:) :: [a] -> a -> [a]
```

```
rev = foldl (<:>) [] where (<:>) = flip (:)
```

Explicatie: Elementele sunt procesate de la stanga la dreapta.

```
rev [x1, ... xn] = (...(([] <:> x1) <:> x2)...) <:> xn
```

Atentie!

Limbajul Kaskell foloseste implicit evaluarea lenesa:

- Expresiile sunt evaluate numai cand este nevoie de valoarea lor.
- Expresiile nu sunt evaluate total elementele care nu sunt folosite raman neevaluate.
- O expresie este evaluate o singura data.

Putem folosi map si filter pe liste infinite.

```
Prelude> inf = map (+10) [1..]
```

```
Prelude> take 3 inf
```

```
[11, 12, 13]
```

In exemplul de mai sus, este acceptata definitia lui inf, fara a fi evaluate. Cand expresia take 3 inf este evaluate, numai primele trei elemente ale lui inf sunt calculate, restul ramanand neevaluate.

```
primes = sieve [2..]
sieve (p:ps) = p : sieve [x | x <- ps, mod x p /= 0]
```

Intuitiv, evaluarea lenesa functioneaza astfel:

```
sieve [2..] -->
2 : sieve [x | x <- [3..], mod x 2 /= 0] -->
2 : sieve (3 : [x | x <- [4..], mod x 2 /= 0]) -->
2 : 3 : sieve ([y | y <- [x | x <- [4..], mod x 2 /= 0], mod y 3 /= 0]) -->
...
```

FOLDR POATE FI FOLOSITA PE LISTE INFINITE (in anumite cazuri), pe cand FOLDL NU POATE FI FOLOSITA PE LISTE INFINITE NICIODATA.

```
Prelude> foldr (*) 0 [1..]
```

Exception: Stack Overflow

|                                                       |                                                       |
|-------------------------------------------------------|-------------------------------------------------------|
| Prelude> take 3 \$ foldr (\x xs -> (x+1):xs) [] [1..] | Prelude> take 3 \$ foldl (\x xs -> (x+1):xs) [] [1..] |
| [2, 3, 4]                                             | Exception: Stack Overflow                             |
| Explicatie: Foldr a functionat pe o lista infinita    | Explicatie: Foldl calculeaza expresia la infinit      |

## TIPURI DE DATE

- Tipuri de date suma
- Tipuri de date produs
- Tipuri de date definite recursiv

### Tipuri de date suma

Bool si Season sunt tipuri de date suma, adica sunt definite prin enumerarea alternativelor.

|                                                                                                                                                              |                                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| data Bool = False   True <ul style="list-style-type: none"> <li>• Bool este constructor de tip</li> <li>• False si True sunt constructori de date</li> </ul> | data Season = Spring   Summer   Autumn   Winter <ul style="list-style-type: none"> <li>• Season este constructor de tip</li> <li>• Spring, Summer, Autumn, Winter sunt constructori de date</li> </ul> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Operatiile se definesc prin pattern matching.

|                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| not :: Bool -> Bool<br>not False = True<br>not True = False<br><br>(&&), (  ) :: Bool -> Bool -> Bool<br>False && q = False<br>True && q = q<br>False    q = q<br>True    q = True | sucesor :: Season -> Season<br>sucesor Spring = Summer<br>sucesor Summer = Autumn<br>sucesor Autumn = Winter<br>sucesor Winter = Spring<br><br>showSeason :: Season -> String<br>showSeason Spring = "Primava ra"<br>showSeason Summer = " Vara"<br>showSeason Autumn = "Toamna"<br>showSeason Winter = "Iarna " |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## Tipuri de date produs

Sa definim u tip de date care sa aiba ca valori punctele cu doua coordonate oarecare.

data Point a b = Pt a b

- Point este constructorul de tip
- Pt este constructor de date

Point este un tip de date produs, definit prin combinarea tipurilor a si b.

Pentru accesarea componentelor, definim proiectiile:

|                                           |                                           |
|-------------------------------------------|-------------------------------------------|
| pr1 :: Point a b -> a<br>pr1 (Pt x _) = x | pr2 :: Point a b -> b<br>pr2 (Pt _ y) = y |
|-------------------------------------------|-------------------------------------------|

Prelude> : t (Pt 1 "c")

(Pt 1 "c") :: Num a => Point a [Char]

Prelude> : t Pt

Pt :: a -> b -> Point a b

-- constructorul de date este operatie

Prelude> : t (Pt 1)

(Pt 1) :: Num a => b -> Point a b

Se pot defini operatii:

pointFlip :: Point a b -> Point b a

pointFlip (Pt x y) = Pt y x

## Tipuri de date definite recursiv

Sa declaram lista ca tip de date algebric:

data Lista = Nil | Cons a (Lista)

- List este constructor de tip
- Nil si Cons sunt constructori de date

Se pot defini operatii:

append :: Lista -> Lista -> Lista

append Nil ys = ys

append (Cons x xs) ys = Cons x (append xs ys)

| Liste cu simboluri                                                                                         | Tupluri cu simboluri                                                                                                                                                                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| data [a] = []   a : [a]<br>Constructorii listelor sunt [] si : unde<br>[] :: [a]<br>(:) :: a -> [a] -> [a] | data (a, b) = (a, b)<br>data (a, b, c) = (a, b, c)<br>Pentru tupluri nu exista o declaratie generica,<br>fiecare declaratie defineste tuplul de lungimea<br>corespunzatoare, iar constructorii pentru fiecare<br>tip în parte sunt:<br>(,) :: a -> b -> (a, b)<br>(,,) :: a -> b -> c -> (a, b, c)<br>... |

|                                                                        |                                                                                                                       |
|------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <pre>wrong :: Int -&gt; Int -&gt; Int wrong n m = divide n m + 3</pre> | <pre>right :: Int -&gt; Int -&gt; Int right n m = case divide n m of     Nothing -&gt; 3     Just r -&gt; r + 3</pre> |
|------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|

## Tipul Either

data Either a b = Left a | Right b

mylist :: [Either Int String]

mylist = [Left 4, Left 1, Right "hello", Left 2, Right " ", Right "world", Left 17]

Definiti o functie care calculeaza suma elementelor întregi.

|                                                                                                                                                |                                                                                              |
|------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <pre>addints :: [Either Int String] -&gt; Int addints [] = 0 addints (Left n : xs) = n + addint s xs addints (Right s : xs) = addints xs</pre> | <pre>addints' :: [Either Int String] -&gt; Int addints' xs = sum [n   Left n &lt;- xs]</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|

Definiti o functie care intoarce concatenarea elementelor de tip String.

|                                                                                                                                                    |                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <pre>addstrs :: [Either Int String] -&gt; String addstrs [] = "" addstrs (Left n : xs) = addstrs xs addstrs (Right s : xs) = s ++ addstrs xs</pre> | <pre>addstrs' :: [Either Int String] -&gt; String addstrs' xs = concat [s   Right s &lt;- xs]</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|

## Utilizarea Type

Cu type se pot redenumi tipuri deja existente.

type FirstName = String

type LastName = String

type Age = Int

type Height = Float

type Phone = String

data Person = Person FirstName LastName Age Height Phone

Datele personale pot fi definite ca inregistrari:

|                                                                                                                                                                                                                       |                                                                                                                                                                                                      |  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| <pre>data Person = Person {firstName :: String,                       lastName :: String,                       age :: Int,                       height :: Float,                       phoneNumber :: String}</pre> | <pre>gigel = Person {firstName = "Gheorghe",                  lastName = "Georgescu",                  age = 10,                  height = 193.5,                  phoneNumber = "0765897543"}</pre> |  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|

Pentru a accesa fiecare element:

- firstName :: Person -> String  
firstName (Person firstname \_ \_ \_ \_) = firstname
- lastName :: Person -> String  
lastName (Person \_ lastname \_ \_ \_ \_) = lastname
- age :: Person -> Int  
age (Person \_ \_ age \_ \_ \_) = age
- height :: Person -> Float  
height (Person \_ \_ \_ height \_) = height
- phoneNumber :: Person -> String  
phoneNumber (Person \_ \_ \_ \_ number) = number



```
Prelude*> let ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"
```

|                           |                        |                             |
|---------------------------|------------------------|-----------------------------|
| Prelude*> firstName ionel | Prelude*> height ionel | Prelude*> phoneNumber ionel |
| "Ion"                     | 175.2                  | "0712334567"                |

Proiectiile sunt definite automat, iar sintaxa specializata pentru actualizari este:

```
nextYear :: Person -> Person
```

```
nextYear person = person {age = age person + 1}
```

Desi toate definitiile sunt corecte, o valoare de tip Person nu poate fi afisata deoarece nu este instanta a clasei Show. Trebuia sa avem:

```
data Person = Person FirstName LastName Age Height Phone deriving Show
```

```
Prelude> nextYear ionel
```

---

No instance for (Show Person) arising from a use of 'print'

## CLASE DE TIPURI

Sa scriem functie my\_elem care testeaza daca un element apartine unei liste.

| Descrieri de liste                   | Recursivitate                                                          | Functii de nivel inalt                              |
|--------------------------------------|------------------------------------------------------------------------|-----------------------------------------------------|
| my_elem x ys = or [x == y   y <- ys] | my_elem x [] = False<br>my_elem x (y : ys) =<br>x == y    my_elem x ys | my_elem x ys = foldr (  ) False<br>(map ( x ==) ys) |

|                              |                                                           |
|------------------------------|-----------------------------------------------------------|
| Prelude> my_elem 1 [2, 3, 4] | Prelude> my_elem (1, 'o') [(0, 'w'), (1, 'o'), (2, 'r') ] |
| False                        | True                                                      |
| Prelude> my_elem 'o' "word"  | Prelude> my_elem "word" ["list", "of", "word"]            |
| True                         | True                                                      |

Functia my\_elem este polimorfica. Definitia unei functii este parametrice in tipul de date. Totusi, definitia nu functioneaza pentru orice tip. De ce?

```
Prelude> my_elem (+ 2) [(+ 2), sqrt]
```

---

No instance for (Eq (Double -> Double)) arising from a use of 'my\_elem'

```
Prelude> :t my_elem
```

---

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

În definitia (my\_elem x ys = or [x == y | y <- ys]) folosim relatia de egalitate == care nu este definita pentru orice tip.

```
Prelude> sqrt == sqrt
```

---

No instance for (Eq (Double -> Double)) ...

```
Prelude> ("ab", 1) == ("ab", 2)
```

---

False

O clasa de tipuri este determinata de o multime de functii (este o interfata).

```
class Eq a where
```

```
    (==) :: a -> a -> Bool
```

```
    (/=) :: a -> a -> Bool      -- minimum definition : (==)
```

```
    x /= y = not (x == y)      -- ^^^ putem avea definitii implicite
```

Puteti verifica folosind comanda :info sau :i ce contine o anumita clasa de tipuri.

Tipurile care apartin clasei sunt instante ale clasei.

```
instance Eq Bool where
```

```
    False == False = True
```

```
    False == True = False
```

```
    True == False = False
```

```
    True == True = True
```

În semnatura functiei my\_elem trebuie sa precizam ca tipul a este în **clasa Eq**.

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

- Eq a se numeste constrângere de tip.
- => separa constrângerile de tip de restul semnaturii.

În exemplul de mai sus am considerat functia my\_elem definita pe liste, dar în realitate e mai complexa:

```
Prelude> :t my_elem
```

---

```
my_elem :: (Eq a, Foldable t) => a -> t a -> Bool
```

În aceasta definitie Foldable este o alta clasa de tipuri, iar t este un parametru care tine locul unui constructor de tip!

```
class Eq a where
```

```
    (==) :: a -> a -> Bool
```

Instantele lui Eq sunt urmatoarele:

- instance Eq Int where  
 (==) = eqInt – built-in
- instance Eq Char where  
 x == y = ord x == ord y
- instance (Eq a, Eq b) => Eq (a, b) where  
 (u, v) == (x, y) = (u == x) && (v == y)
- instance Eq a => Eq [a] where  
 [] == [] = True  
 [] == y : ys = False  
 x : xs == [] = False  
 x : xs == y : ys = (x == y) && (xs == ys)

Clasele pot fi extinse.

```
class (Eq a) => Ord a where
```

```
    (<) :: a -> a -> Bool
```

```
    (<=) :: a -> a -> Bool
```

```
    (>) :: a -> a -> Bool
```

```
    (>=) :: a -> a -> Bool
```

```
    -- minimum definition: (<=)
```

```

x < y = x <= y && x /= y
x > y = y < x
x >= y = y <= x

```

**Clasa Ord** este clasa tipurilor de date înzestrate cu o relatie de ordine. În definitia clasei Ord s-a impus o constrângere de tip. Astfel, orice instanta a clasei Ord trebuie sa fie instanta a clasei Eq.

Instancele lui Ord sunt urmatoarele:

- instance Ord Bool where  
False <= False = True  
False <= True = True  
True <= False = False  
True <= True = True
- instance (Ord a, Ord b) => Ord (a, b) where  
(x, y) <= (x', y') = x < x' || (x == x' && y <= y') -- ordinea lexicografica
- instance Ord a => Ord [a] where  
[] <= ys = True  
(x : xs) <= [] = False  
(x : xs) <= (y : ys) = x < y || (x == y && xs <= ys)

### Definirea claselor

Sa presupunem ca vrem sa definim o clasa de tipuri pentru datele care pot fi afisate. O astfel de clasa trebuie sa contina o metoda care sa indice modul de afisare:

```
class Visible a where
```

```
  toString :: a -> String
```

Putem face instantieri astfel:

```
instance Visible Char where
```

```
  toString c = [c]
```

Clasele Eq, Ord sunt predefinite. Clasa Visible este definita de noi, dar exista o clasa predefinita care are acelasi rol: **clasa Show**.

```
class Show a where
```

```
  show :: a -> String -- analogul lui "toString"
```

Instancele lui Show sunt urmatoarele:

- 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

Constructorii simboluri:

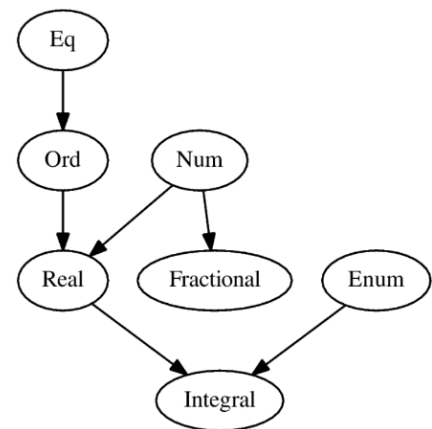
```
data List a = Nil | a ::: List a
```

```
infixr 5 :::
```

|                                                                                                                                                                                                                            |                                                                                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>eqList :: Eq a =&gt; List a -&gt; List a -&gt; Bool eqList Nil Nil = True eqList (x :: xs)(y :: ys) = x == y &amp;&amp; eqList xs ys eqList _ = False instance (Eq a) =&gt; Eq (List a) where     (==) = eqList</pre> | <pre>showMyList :: Show a =&gt; List a -&gt; String showMyList Nil = "Nil" showMyList (x :: xs) = show x ++ " :: " ++ showMyList xs  instance (Show a) =&gt; Show (List a) where     show = showMyList</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Clase de tipuri pentru numere:

- class (Eq a, Show a) => Num a where  
 (+), (-), (\*) :: a -> a -> a  
 negate :: a -> a  
 ...  
 fromInteger :: Integer -> a  
 -- minimum definition: (+), (-), (\*), fromInteger  
 negate x = fromInteger 0 - x
- class (Num a) => Fractional a where  
 (/) :: a -> a -> a  
 recip :: a -> a  
 fromRational :: Rational -> a  
 ...  
 -- minimum definition : (/), fromRational  
 recip x = 1/x
- class (Num a, Ord a) => Real a where  
 toRational :: a -> Rational  
 ...
- class (Real a, Enum a) => Integral a where  
 div, mod :: a -> a -> a  
 toInteger :: a -> Integer  
 ...



#### Derivarea automata pentru tipurile algebrice

|                                                                                         |                                                                 |
|-----------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| <pre>data Season = Spring   Summer   Autumn   Winter     deriving (Eq, Ord, Show)</pre> | <pre>data Point a b = Pt a b     deriving (Eq, Ord, Show)</pre> |
|-----------------------------------------------------------------------------------------|-----------------------------------------------------------------|

Cum putem sa le facem instante ale claselor Eq, Ord, Show? Putem sa le facem explicit sau sa folosim derivarea automata. Derivarea automata poate fi folosita numai pentru unele clase predefinite.

| Instantierea prin derivare automata                | Instantierea explicita                                                                          |
|----------------------------------------------------|-------------------------------------------------------------------------------------------------|
| <pre>data Point a b = Pt a b     deriving Eq</pre> | <pre>instance Eq a =&gt; Eq (Point a b) where     (==) (Pt x1 y1) (Pt x2 y2) = (x1 == x2)</pre> |

Egalitatea, relatia de ordine si modalitatea de afisare sunt definite implicit daca este posibil:

|                                                                                                                              |                                                     |
|------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| <pre>Prelude&gt; Pt 2 3 &lt; Pt 5 6 True</pre>                                                                               | <pre>Prelude&gt; Pt 2 "b" &lt; Pt 2 "a" False</pre> |
| <pre>Prelude&gt; Pt (+ 2) 3 &lt; Pt (+ 5) 6 No instance for (Ord (Integer -&gt; Integer)) arising from a use of '&lt;'</pre> |                                                     |