

Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Ana Cristina Turlea

Departamentul de Informatică, FMI, UB
ioana@fmi.unibuc.ro
ana.turlea@fmi.unibuc.ro

1 Elemente de sintaxă

2 Legarea variabilelor

3 Tipuri de date

4 Funcții

5 Liste

Elemente de sintaxă

Sintaxă

- Comentarii

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

- Identificatori

- șiruri formate din litere, cifre, caracterele `_` și `'` (apostrof)
- identificatorii pentru variabile încep cu literă mică sau `_`
- identificatorii pentru tipuri și constructori încep cu literă mare
- Haskell este sensibil la majuscule (case sensitive)

```
double x = 2 * x
data Point a = Pt a a
```

Sintaxă

Blocuri și indentare

Blocurile sunt delimitate prin indentare.

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

Sintaxă

Blocuri și indentare

Blocurile sunt delimitate prin indentare.

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

```
trei =  let
         a = 1
         b = 2
       in a + b
```

Sintaxă

Blocuri și indentare

Blocurile sunt delimitate prin indentare.

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

```
trei =  let
        a = 1
        b = 2
      in a + b
```

- echivalent, putem scrie

```
trei = let {a = 1; b = 2} in a + b
trei = let a = 1; b = 2 in a + b
```

Legarea variabilelor

Variabile

Presupunem că fisierul `test.hs` conține

```
x=1
```

```
x=2
```

- Ce valoare are `x`?

Variabile

Presupunem că fisierul `test.hs` conține

```
x=1
x=2
```

- Ce valoare are `x`?

```
Prelude> :l test.hs
```

```
test.hs:2:1: error:
  Multiple declarations of 'x'
  Declared at: test.hs:1:1
               test.hs:2:1
```

```
2 | x=2
  | ^
```

Variabile

În Haskell, variabilele sunt imuabile, adică:

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

Legarea variabilelor

let .. in ...

este o *expresie* care crează scop local

Presupunem că fișierul testlet.hs conține

```
x=1
```

```
z= let x=3 in x
```

```
Prelude> :l testlet.hs  
[1 of 1] Compiling Main  
Ok, 1 module loaded.
```

```
*Main> z
```

```
3
```

```
*Main> x
```

```
1
```

Legarea variabilelor

- **let .. in ...** crează scop local

```
x = let
    z = 5
    g u = z + u
in let
    z = 7
in g 0 + z
```

Legarea variabilelor

- **let .. in ...** crează scop local

```

x = let
    z = 5
    g u = z + u
in let
    z = 7
    in g 0 + z
-- x=12

```

Legarea variabilelor

- **let .. in ...** crează scop local

```

x = let
    z = 5
    g u = z + u
in let
    z = 7
    in g 0 + z
-- x=12

```

```

x = let z = 5; g u = z + u in let z = 7 in g 0

```

Legarea variabilelor

- **let .. in ...** crează scop local

```

x = let
    z = 5
    g u = z + u
in let
    z = 7
    in g 0 + z
-- x=12

```

```

x = let z = 5; g u = z + u in let z = 7 in g 0 -- x=5

```


Legarea variabilelor

- **let .. in ...** crează scop local

```

x = let
    z = 5
    g u = z + u
in let
    z = 7
    in g 0 + z
-- x=12

```

```

x = let z = 5; g u = z + u in let z = 7 in g 0 -- x=5

```

- clauza ... **where** ... creaza scop local

```

f x = g x + g x + z
where
    g x = 2*x
    z = x-1

```

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 de date

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

Sistemul tipurilor

Tipurile de baza

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

- tipuri compuse: tupluri si liste

```
Prelude> :t ('a', True)  
('a', True) :: (Char, Bool)
```

```
Prelude> :t ["ana", "ion"]  
["ana", "ion"] :: [[Char]]
```

Sistemul tipurilor

Tipurile de baza

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

- tipuri compuse: tupluri si liste

```
Prelude> :t ('a', True)  
('a', True) :: (Char, Bool)
```

```
Prelude> :t ["ana", "ion"]  
["ana", "ion"] :: [[Char]]
```

- tipuri noi definite de utilizator

```
data RGB = Rosu | Verde | Albastru  
data Point a = Pt a a      -- tip parametrizat  
                           -- a este variabila de tip
```

Tipuri de date

- **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 date

- **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"]
```

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]
```

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim în GHCi dacă introducem comanda

```
Prelude> :t 1
```

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim în 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
```

```
Prelude> :t [1,2,3]
```

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

Funcții

Funcții în Haskell. Terminologie

Prototipul funcției

- **numele funcției**
- **signatura funcției**

double :: Integer -> Integer

Definiția funcției

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

double **elem** = elem + elem

Aplicarea funcției

- **numele funcției**
- **parametrul actual (argumentul)**

double **5**

Exemplu: funcție cu două argumente

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**

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 (elem1, elem2)

- **numele funcției**
- **argumentul**

Tipuri de funcții

Prelude> :t abs

abs :: Num a => a -> a

Prelude> :t div

div :: Integral a => a -> a -> a

Prelude> :t (:)

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

Prelude> :t (++)

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

Prelude> :t zip

zip :: [a] -> [b] -> [(a, b)]

Definirea funcțiilor

```
fact :: Integer -> Integer
```

- Definiție folosind **if**

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

Definirea funcțiilor

`fact :: Integer -> Integer`

- Definiție folosind **if**

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

- Definiție folosind ecuații

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

Definirea funcțiilor

```
fact :: Integer -> Integer
```

- Definiție folosind **if**

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

- Definiție folosind ecuații

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

- Definiție folosind cazuri

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

Șabloane (patterns)

- $x:y = [1,2,3] \text{ -- } x=1 \text{ si } y=[2,3]$

Observați că : este constructorul pentru liste.

Șabloane (patterns)

- $x:y = [1,2,3] \text{ -- } x=1 \text{ si } y=[2,3]$

Observați că : este constructorul pentru liste.

- $(u,v)=('a' ,[(1, 'a') ,(2, 'b')]) \text{ -- } u='a' ,$
 $\text{ -- } v=[(1, 'a') ,(2, 'b')]$

Observați că (,,) este constructorul pentru tupluri.

Șabloane (patterns)

- $x:y = [1,2,3]$ -- $x=1$ si $y=[2,3]$

Observați că : este constructorul pentru liste.

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

Observați că (,,) este constructorul pentru tupluri.

- Definiții 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

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]`

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 semnatura 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]`

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 semnatura 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]

Liste

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 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]
```

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 $\{x \mid P(x)\}$

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

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

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

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

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

```
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]
```


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

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

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]]

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

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

```
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)]
```

```
Prelude> let xs = ['A'..'Z']
```

```
Prelude> [x | (i,x) <- [1..] 'zip' xs, even i]
```

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

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

```
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)]
```

```
Prelude> let xs = ['A'..'Z']
```

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

zip xs ys

```
Prelude> let xs = ['A'..'Z']
```

```
Prelude> [x | (i,x) <- [1..] 'zip' xs, even i]
```

zip xs ys

```
Prelude> let xs = ['A'..'Z']
```

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

```
Prelude> :t zip
```

```
zip :: [a] -> [b] -> [(a, b)]
```

```
Prelude> let ys = ['A'..'E']
```

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

zip xs ys

```
Prelude> let xs = ['A'.. 'Z']
```

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

```
Prelude> :t zip
```

```
zip :: [a] -> [b] -> [(a, b)]
```

```
Prelude> let ys = ['A'.. 'E']
```

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

Observați diferența!

```
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')]
```

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 leneșe**, se pot defini liste infinite (fluxuri de date)

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


Liste infinite

Drept consecință a **evaluării leneșe**, 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]
```

Liste infinite

Drept consecință a **evaluării leneșe**, 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)]
```

Pe săptămâna viitoare!