

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C02

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Funcții

Funcții în Haskell. Terminologie

Prototipul funcției

- numele funcției
- semnătura 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
- semnătura 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
- semnatura funcției

Definiția funcției

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

- numele funcției
- parametrul formal
- corpul funcției

Aplicarea funcției

dist (5, 7)

- numele funcției
- argumentul

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)
```

Definirea funcțiilor folosind șabloane și ecuații

```
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 apelată se încearcă potrivirea 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

In Haskell, ordinea ecuațiilor este importanta.

Să presupunem că schimbăm ordinea ecuațiilor din definiția factorialului:

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

Ce se întâmplă?

Definirea funcțiilor folosind șabloane și ecuații

In Haskell, ordinea ecuațiilor este importanta.

Să presupunem că schimbăm ordinea 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**.

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.

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]

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZu3yb>

Seria 24: <https://questionpro.com/t/AT4NiZu3J8>

Seria 25: <https://questionpro.com/t/AT4qgZu3yd>

Liste

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

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

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> 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)]
```

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

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

Putem folosi **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> :t zip
```

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

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

```
Prelude> zip [1..] ys
```

```
[(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'), (5, 'E')]
```

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

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

```
"BDFHJLNPRTVXZ"
```

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

Liste infinite

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

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

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

```
Prelude> evenNat = [0,2..] -- progresie infinita
```

```
Prelude> take 7 evenNat  
[0,2,4,6,8,10,12]
```

```
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)]
```

Șabloane (patterns) pentru liste

Listele sunt construite folosind constructorii (:) și []

`[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]

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

$$(++) :: [a] \rightarrow [a] \rightarrow [a],$$

dar `[x] ++ [1] = [2,1]` **nu** va avea ca efect legarea lui `x` la `2`.

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

```
Prelude> x
```

error: ...

Șabloanele sunt liniare

Sabloanele sunt **liniare**, adică o variabilă apare cel mult odată.

Șabloane în care o variabilă apare de mai multe ori provoacă 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 soluție este folosirea gărzilor:

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

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

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZu3yi>

Seria 24: <https://questionpro.com/t/AT4NiZu3J9>

Seria 25: <https://questionpro.com/t/AT4qgZu3yp>

Pe săptămâna viitoare!