Programare Funcțională

Funcții și liste în Haskell

Ioana Leuștean
Traian Florin Șerbănuță

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

Funcții

Funcții în Haskell. Terminologie

Prototipul funcției

double :: Integer -> Integer

- **numele funcției**
- signatura funcției

Definiția funcției

double elem = elem + elem

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

Aplicarea funcției

double 5

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

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

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]

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]

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]
```

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?

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 nici un argument de intrare.

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 pentru tupluri

Observați că (,) este constructorul pentru perechi.

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

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
```

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

```
Prelude> xs = [0..10]
Prelude> [x | x <- xs, even x]
```

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

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

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

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

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)| x <- [x1,. . . ,xn], 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)]
```

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

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

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

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

zip xs ys

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

zip xs ys

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

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')]
```

zip xs ys

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

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')]
```

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> [head [],3] !! 1
*** Exception: Prelude.head: empty list
```

Liste infinite

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

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

Liste infinite

Drept consecință a evaluării lenești, 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]
```

Liste infinite

Drept consecință a evaluării lenește, 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:[]
```

Observați:

```
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

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
error: ...
```

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
```

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
```

error: Conflicting definitions for x

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
```

error: Conflicting definitions for x

O soluție este folosirea gărzilor:

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

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

Observație: Se pot combina ecuații cu gărzi cu ecuații fără.

Pe săptămâna viitoare!