Programare funcțională

Liste și funcții în Haskell

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Liste

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 :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 :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
```

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
```

String = listă de caractere

- String: "prog\nfunc"

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

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

Prelude> lines "prog\nfunc"
["prog","func"]
Prelude> words "pr og\nfu nc"
["pr","og","fu","nc"]
```

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

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

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

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

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

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

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

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

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

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

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

Folosirea lui `let` pentru declarații locale:

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

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

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

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

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

Folosirea lui `let` pentru declarații locale:

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

Prelude> let 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> 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)]
```

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: 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 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

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

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

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

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 valori pozitive.

Definirea funcțiilor folosind șabloane și ecuații

Tipul Bool este definit în Haskell astfel:

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

Putem defini operația || astfel

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

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

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

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

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

Șabloane 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')]
```

- 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
```

Șabloane (patterns) pentru liste

Listele sunt construite folosind constructorii (:) și []

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

Observați:

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

Ce s-a întâmplat?

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

Șabloane (patterns) pentru liste

Definiții folosind șabloane

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

- x:xs se potrivește cu liste nevide

Şabloane (patterns) pentru liste

Definiții folosind șabloane

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

- x:xs se potrivește cu liste nevide

Atenție!

Şabloanele 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

```haskell
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:

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

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

Currying

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.

- Pentru $x \in A$ (arbitrar, fixat) definim

  $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.

  Funcția f_x se obține prin aplicarea parțială a funcției f.

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.

- Pentru $x \in A$ (arbitrar, fixat) definim

  $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.

  Funcția f_x se obține prin aplicarea parțială a funcției f.

  In mod similar definim *aplicarea parțială* pentru orice $y \in B$

  $f^y : A \to C$, $f^y(x) = z$ dacă și numai dacă $f(x, y) = z$.

Funcții în matematică

Exemplu

A =Int, $B = C$ =String

$$f(x,y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

- Fie $x \in$ Int arbitrar, fixat. Atunci f_x :String$\to$String și
  - dacă $x <= 0$, atunci $f_x(y) = ""$ oricare y
  - dacă $x > 0$ atunci $f_x(y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \end{cases}$

Funcții în matematică

Exemplu

A =Int, $B = C$ =String

$$f(x,y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

- Fie $x \in$ Int arbitrar, fixat. Atunci f_x :String→String și
  - dacă $x <= 0$, atunci $f_x(y) = ""$ oricare y
  - dacă $x > 0$ atunci $f_x(y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \end{cases}$
- Fie $y \in$String arbitrar, fixat. Atunci f^y :Int→String și
  $$f^y(x) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.
- Dacă notăm $B \to C \stackrel{not}{=} \{h : B \to C \mid h \text{ funcție}\}$ observăm că $f_x \in B \to C$ pentru orice $x \in A$.

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.
- Dacă notăm $B \to C \stackrel{not}{=} \{h : B \to C \mid h \text{ funcție}\}$ observăm că $f_x \in B \to C$ pentru orice $x \in A$.
- Asociem lui f funcția

  $cf : A \to (B \to C), \;\; cf(x) = f_x$

  Observăm că pentru fiecare element $x \in A$, funcția cf întoarce ca rezultat funcția $f_x \in B \to C$, adică

  $cf(x)(y) = z$ dacă și numai dacă $f(x, y) = z$

Funcții în matematică

- Fie $f : A \times B \to C$ o funcție. În mod uzual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \to C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$.
- Dacă notăm $B \to C \stackrel{not}{=} \{h : B \to C \mid h \text{ funcție}\}$ observăm că $f_x \in B \to C$ pentru orice $x \in A$.
- Asociem lui f funcția

  $$cf : A \to (B \to C), \;\; cf(x) = f_x$$

  Observăm că pentru fiecare element $x \in A$, funcția cf întoarce ca rezultat funcția $f_x \in B \to C$, adică

  $$cf(x)(y) = z \text{ dacă și numai dacă } f(x, y) = z$$

Forma curry

Vom spune că funcția cf este *forma curry* a funcției f.

De la matematică la Haskell

Funcția $f : \text{Int} \times \text{String} \to \text{String}$

$$f(x, y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

poate fi definită în Haskell astfel:

```haskell
f :: (Int, String) -> String
f (n,s) = take n s
```

De la matematică la Haskell

Funcția f : Int $\times$ String $\to$ String

$$f(x,y) = \begin{cases} z, & |y| >= x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x <= 0 \end{cases}$$

poate fi definită în Haskell astfel:

```haskell
f :: (Int, String) -> String
f (n,s) = take n s
```

Observăm că:

```
Prelude> let cf = curry f
Prelude> :t cf
cf :: Int -> String -> String
Prelude> f(1,"abc")
"a"
Prelude> cf 1 "abc"
"a"
```

Currying

"Currying" este procedeul prin care o funcție cu mai multe argumente este transformată într-o funcție care are un singur argument și întoarce o altă funcție.

- In Haskell toate funcțiile sunt forma **curry**, deci au un singur argument.
- Operatorul $\rightarrow$ pe tipuri este asociativ la dreapta, adică tipul $a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n$ îl gândim ca $a_1 \rightarrow (a_2 \rightarrow \cdots (a_{n-1} \rightarrow a_n) \cdots)$.
- Aplicarea funcțiilor este asociativă la stânga, adică expresia $f\ x_1 \cdots x_n$ o gândim ca $(\cdots((f\ x_1)\ x_2) \cdots x_n)$.

Funcții și mulțimi

Teoremă

Mulțimile $(A \times B) \to C$ și $A \to (B \to C)$ sunt echipotente.

Funcții și mulțimi

Teoremă

Mulțimile $(A \times B) \to C$ și $A \to (B \to C)$ sunt echipotente.

Observație

Funcțiile curry și uncurry din Haskell stabilesc bijecția din teoremă:

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

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]
```

Funcții anonime

Funcții anonime = lambda expresii

\x1 x2 $\cdots$ xn -> expresie

Funcții anonime

Funcții anonime = lambda expresii

\x1 x2 $\cdots$ xn -> expresie

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

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

- Funcțiile sunt valori (first-class citizens)
  - pot fi folosite ca argumente pentru alte funcții

λ-calcul (pe scurt)

Structura λ-expresiilor

O expresie este definită recursiv astfel:

- este o variabilă (un identificator)
  x
- se obține prin abstractizarea unei variabile x într-o altă expresie e
  $\lambda x.e$ exemplu: $\lambda x.x$
- se obține prin aplicarea unei expresii e_1 asupra alteia e_2
  $e_1\ e_2$ exemplu: $(\lambda x.x)y$

Operația de abstractizare $\lambda x.e$

- reprezintă o funcție anonimă
- constă din două parți: antetul $\lambda x.$ și corpul e
- variabila x din anter este parametrul funcției
  - leagă aparițiile variabilei x în e (ca un cuantificator)
  - Exemplu: $\lambda x.xy$ — x e legată, y e liberă
- Corpul funcției reprezintă expresia care definește funcția

α-echivalență

- Redenumirea unui argument și a tuturor aparițiilor sale legate
  - Exemplu: $\lambda x.x \equiv_\alpha \lambda y.y \equiv_\alpha \lambda a.a$
  - Asemanator cu: $f(x) = x$ vs $f(y) = y$ vs $f(a) = a$
- Numele asociat argumentului e pur formal
  - E necesar doar ca să îl pot recunoaște în corpul funcției
  - Există reprezentări fără argumente (e.g. indecși de Bruijn)
- α-echivalența redenumește doar aparițiile legate ale argumentului
  - Exemple:
    $(\lambda x.x)x \not\equiv_\alpha (\lambda y.y)y$
    $(\lambda x.x)x \equiv_\alpha (\lambda y.y)x$

β-reducție

Cum aplicăm o funcție (anonimă) unui argument?

înlocuim aplicația cu corpul funcției în care substituim aparițiile variabilei legate cu argumentul dat.

$$(\lambda x.e)e' \to_\beta e[x := e']$$

Exemple

$$(\lambda x.x)y \to_\beta x[x := y] = y$$

$$(\lambda x.x\ x)\lambda x.x \to_\beta x\ x[x := \lambda x.x] = (\lambda x.x)\lambda x.x \to_\beta x[x := \lambda x.x] = \lambda x.x$$

β-reducție — alte exemple

Aplicarea funcțiilor se grupează la stânga

$(\lambda x.x)(\lambda y.y)z = ((\lambda x.x)(\lambda y.y))z$

β-reducție — alte exemple

Aplicarea funcțiilor se grupează la stânga

$$(\lambda x.x)(\lambda y.y)z = ((\lambda x.x)(\lambda y.y))z$$
$$((\lambda x.x)(\lambda y.y))z \rightarrow_\beta (x[x := \lambda y.y])z = (\lambda y.y)z$$
$$(\lambda y.y)z \rightarrow_\beta y[y := z] = z$$

Funcție cu variabile libere

$$(\lambda x.x\ y)z \rightarrow_\beta (x\ y[x := z]) = z\ y$$

lambda are are prioritate foarte mică

$$\lambda x.x\ \lambda x.x = \lambda x.(x\ (\lambda x.x))$$

Mai multe argumente

- Funcțiile anonime au un singur parametru
  - și pot fi aplicate unui singur argument
- Simulăm mai multe argumente prin abstractizare repetată
  Exemplu: $\lambda x.\lambda y.x\ y$
  - Citim: primește ca argumente x și y și aplică pe x lui y
  - De fapt e o funcție de x care în urma aplicării întoarce cadă o funcție de y
  - Pentru simplificarea notației, scriem $\lambda x\ y.x\ y$ în loc de $\lambda x.\lambda y.x\ y$

Exemplu de evaluare

$$(\lambda x\ y.x\ y)(\lambda z.a)1 = (\lambda x.(\lambda y.x\ y))(\lambda z.a)1 = ((\lambda x.(\lambda y.x\ y))(\lambda z.a))1$$

Mai multe argumente

- Funcțiile anonime au un singur parametru
  - și pot fi aplicate unui singur argument
- Simulăm mai multe argumente prin abstractizare repetată
  Exemplu: $\lambda x.\lambda y.x\ y$
  - Citim: primește ca argumente x și y și aplică pe x lui y
  - De fapt e o funcție de x care în urma aplicării întoarce cadă o funcție de y
  - Pentru simplificarea notației, scriem $\lambda x\ y.x\ y$ în loc de $\lambda x.\lambda y.x\ y$

Exemplu de evaluare

$$(\lambda x\ y.x\ y)(\lambda z.a)1 = (\lambda x.(\lambda y.x\ y))(\lambda z.a)1 = ((\lambda x.(\lambda y.x\ y))(\lambda z.a))1 \rightarrow_\beta$$
$$((\lambda y.x\ y)[x := \lambda z.a])1 = (\lambda y.(\lambda z.a)y)1$$

Mai multe argumente

- Funcțiile anonime au un singur parametru
  - și pot fi aplicate unui singur argument
- Simulăm mai multe argumente prin abstractizare repetată
  Exemplu: $\lambda x.\lambda y.x\ y$
  - Citim: primește ca argumente x și y și aplică pe x lui y
  - De fapt e o funcție de x care în urma aplicării întoarce cadă o funcție de y
  - Pentru simplificarea notației, scriem $\lambda x\ y.x\ y$ în loc de $\lambda x.\lambda y.x\ y$

Exemplu de evaluare

$(\lambda x\ y.x\ y)(\lambda z.a)1 = (\lambda x.(\lambda y.x\ y))(\lambda z.a)1 = ((\lambda x.(\lambda y.x\ y))(\lambda z.a))1 \rightarrow_\beta ((\lambda y.x\ y)[x := \lambda z.a])1 = (\lambda y.(\lambda z.a)y)1 \rightarrow_\beta ((\lambda z.a)y)[y := 1] = (\lambda z.a)1$

Mai multe argumente

- Funcțiile anonime au un singur parametru
  - și pot fi aplicate unui singur argument
- Simulăm mai multe argumente prin abstractizare repetată
  Exemplu: $\lambda x.\lambda y.x\ y$
  - Citim: primește ca argumente x și y și aplică pe x lui y
  - De fapt e o funcție de x care în urma aplicării întoarce cadă o funcție de y
  - Pentru simplificarea notației, scriem $\lambda x\ y.x\ y$ în loc de $\lambda x.\lambda y.x\ y$

Exemplu de evaluare

$(\lambda x\ y.x\ y)(\lambda z.a)1 = (\lambda x.(\lambda y.x\ y))(\lambda z.a)1 = ((\lambda x.(\lambda y.x\ y))(\lambda z.a))1 \rightarrow_\beta$
$((\lambda y.x\ y)[x := \lambda z.a])1 = (\lambda y.(\lambda z.a)y)1 \rightarrow_\beta ((\lambda z.a)y)[y := 1] =$
$(\lambda z.a)1 \rightarrow_\beta a[z := y] = a$

Programarea funcțională

- Paradigmă de programare ce folosește funcții modelate după funcțiile din matematică.
- Programele se obțin ca o combinație de expresii.
- Expresiile pot fi valori concrete, variable și funcții.
- Funcțiile sunt expresii ce pot fi aplicate unor intrări.
  - În urma aplicării, o funcție e redusă sau evaluată.
- Funcțiile sunt valori (first-class citizens)
  - pot fi folosite ca argumente pentru alte funcții

Pe săptămâna viitoare!