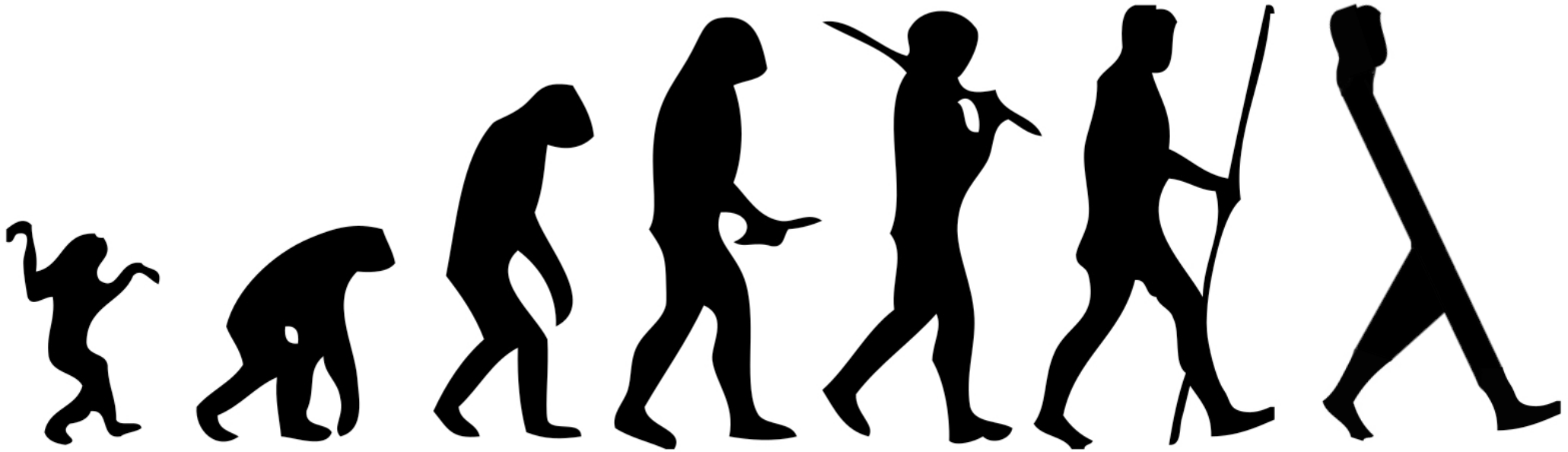
PARADIGME DE PROGRAMARE

Curs 6

Limbajul Haskell. Tipare tare / slabă / statică / dinamică. Tipuri și expresii de tip.

Programare funcțională în Haskell



Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii `.` și `$`
- Expresii condiționale
- Legare statică
- Evaluare lenesă

Sintaxa Haskell

- **Notație infixată pentru operatori**
  `1 + 2, a < 5`
- **Notație prefixată pentru funcții**
  `filter odd [1 .. 7], f 2 + f 5`
- **Parantezele se folosesc pentru controlul priorității (nu pentru aplicația de funcție)**
  `f (1 + 2), f (g x)`
  `(f 1) + 2` este echivalent cu `f 1 + 2`, întrucât aplicația de funcție are cea mai mare prioritate
  `(f g) x` este echivalent cu `f g x`, întrucât aplicația de funcție este asociativă la stânga
- **Indentarea înlocuiește controlul prin separatori ca { } sau ;**
  Orice cod din **corpul** unei expresii trebuie indentat **mai la dreapta** decât începutul expresiei!
  O **nouă expresie** începe pe **același nivel sau mai la stânga** față de începutul expresiei anterioare!

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii `.` și `$`
- Expresii condiționale
- Legare statică
- Evaluare lenesă

TDA-ul Pereche

Constructori de bază

(,) : T1 x T2 -> Pereche // creează o pereche între orice 2 argumente

Operatori

fst : Pereche -> T1 // extrage prima valoare din pereche

snd : Pereche -> T2 // extrage a doua valoare din pereche

Exemple

```
(1, "unu")
a = (('a', 2), "a2")
fst a
snd (fst a)
```

TDA-ul Pereche

Constructori de bază

(,) : T1 x T2 -> Pereche // creează o pereche între orice 2 argumente

Operatori

fst : Pereche -> T1 // extrage prima valoare din pereche

snd : Pereche -> T2 // extrage a doua valoare din pereche

Exemple

```
(1, "unu")                    -- (1, "unu")
a = (('a', 2), "a2")
fst a                         -- ('a', 2)
snd (fst a)                   -- 2
```

TDA-ul Listă

Constructori (de bază și nu numai)

[ ] : -> Listă // creează o listă vidă

: : T x Listă -> Listă // creează o listă prin adăugarea unei valori la începutul unei liste

[, , .. ,] : T x .. T -> Listă // creează o listă din toate argumentele sale (de același tip)

Operatori

head : Listă -> T

tail : Listă -> Listă

null : Listă -> Bool

length : Listă -> Nat

++ : Listă x Listă -> Listă

Exemple

```
head [[2,4],[6],[5]]
tail (2:3:[4,5])
null [[]]
length [[]]
[1] ++ [1,2,3] ++ [4,5]
```

TDA-ul Listă

Constructori (de bază și nu numai)

[ ] : -> Listă // creează o listă vidă

: : T x Listă -> Listă // creează o listă prin adăugarea unei valori la începutul unei liste

[, , .. ,] : T x .. T -> Listă // creează o listă din toate argumentele sale (de același tip)

Operatori

head : Listă -> T

tail : Listă -> Listă

null : Listă -> Bool

length : Listă -> Nat

++ : Listă x Listă -> Listă

Exemple

```
head [[2,4],[6],[5]]      -- [2,4]
tail (2:3:[4,5])          -- [3,4,5]
null [[]]                 -- False
length [[]]               -- 1
[1] ++ [1,2,3] ++ [4,5]   -- [1,1,2,3,4,5]
```

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii `.` și `$`
- Expresii condiționale
- Legare statică
- Evaluare lenesă

Funcții anonime în Haskell

```
\parametri -> corp
```

Exemple

λx.x	`\x -> x`	
λx.λy.(x y)	`\x -> \y -> x y`	echivalent cu
	`\x y -> x y`	(întrucât funcțiile Haskell sunt **automat curry**)
(λx.x λx.y)	`(\x -> x) (\x -> y)`	

Funcții cu nume în Haskell

```
f parametri = corp
```

Exemple

```
arithmeticMean = \x -> \y -> (x + y)/2
arithmeticMean = \x y -> (x + y)/2
arithmeticMean x y = (x + y)/2

f = arithmeticMean 3        -- se creează \y -> (3 + y)/2
f 18                        -- 10.5
```

echivalent cu

și cu

Simularea funcțiilor uncurry

Definițiile de tipul

`f` x_1 x_2 `...` x_n `= corp`

generează funcții **curry**, care pot fi aplicate pe oricâți (≤n) parametri la un moment dat.

`f` e_1 e_2 `...` e_k întoarce o nouă funcție `\`x_{k+1} `...` x_n `-> corp`$_{[ei/xi]}$.

Pentru a obține comportamentul de funcție **uncurry**, parametrul lui f trebuie să fie un tuplu:

`f (`x_1`,` x_2 `...,` x_n`)= corp`

Exemplu

```
arithmeticMean (x,y) = (x + y)/2
arithmeticMean (3,18)
```

Transformări operator – funcție

- **(op) face transformarea operator → funcție**

```
(-) 3 5                    -- -2
(||) (1<2) (5<3)           -- True
foldr (+) 0 [1..5]         -- 15
(/=) 2 2                   -- False
```

- **`f` face transformarea funcție → operator**

```
5 `mod` 3                  -- 2
(div 6) `map` [1,2,3]      -- [6,3,2]
((==) 2) `filter` [1,2,3]  -- [2]
```

Secțiuni (aplicare parțială a operatorilor)

- **Când se dă operandul din stânga, se așteaptă operandul din dreapta**

```
(5/) 2
map (2-) [0..4]
filter (2<) [0..4]
```

- **Când se dă operandul din dreapta, se așteaptă operandul din stânga**

```
(/5) 2
map (-2) [0..4]
map (/2) [0..4]
filter (<2) [0..4]
```

Secțiuni (aplicare parțială a operatorilor)

- **Când se dă operandul din stânga, se așteaptă operandul din dreapta**

```
(5/) 2                  -- 2.5
map (2-) [0..4]         -- [2,1,0,-1,-2]
filter (2<) [0..4] -- [3,4]
```

- **Când se dă operandul din dreapta, se așteaptă operandul din stânga**

```
(/5) 2
map (-2) [0..4]
map (/2) [0..4]
filter (<2) [0..4]
```

Secțiuni (aplicare parțială a operatorilor)

- **Când se dă operandul din stânga, se așteaptă operandul din dreapta**

```
(5/) 2                  -- 2.5
map (2-) [0..4]         -- [2,1,0,-1,-2]
filter (2<) [0..4] -- [3,4]
```

- **Când se dă operandul din dreapta, se așteaptă operandul din stânga**

```
(/5) 2                  -- 0.4
map (-2) [0..4]         -- eroare, aici -2 e număr, nu funcție
map (/2) [0..4]         -- [0.0,0.5,1.0,1.5,2.0]
filter (<2) [0..4] -- [0,1]
```

Limbajul Haskell – Cuprins

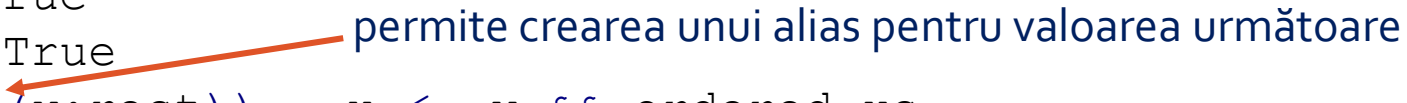
- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii `.` și `$`
- Expresii condiționale
- Legare statică
- Evaluare lenesă

Definirea funcțiilor prin pattern matching

Se descrie comportamentul funcțiilor în funcție de structura parametrilor – ca la scrierea axiomelor TDA-ului.

Exemple

```
1.  fib 0 = 0
2.  fib 1 = 1
3.  fib n = fib (n-2) + fib (n-1)
4.
5.  sumL [] = 0
6.  sumL (x:xs) = x + sumL xs
7.
8.  sumP (x,y) = x + y
9.
10. ordered [] = True
11. ordered [x] = True
12. ordered (x:xs@(y:rest)) = x <= y && ordered xs
```



La aplicare

```
1.  fib 0 = 0
2.  fib 1 = 1
3.  fib n = fib (n-2) + fib (n-1)

fib 3
```

- Dacă argumentul se potrivește cu parametrul din primul punct (primul pattern)
  - Se folosește definiția din primul punct
  - Se ignoră definițiile următoare
- Altfel
  - Se încearcă potrivirea cu punctul următor, ș.a.m.d.

Consecință: Ordinea contează!

Pattern-uri exhaustive

Este important să specificăm comportamentul funcției pe toate valorile tipului – ca la scrierea axiomelor TDA-ului.

```
1.  ordered [] = True
2.  ordered [x] = True
3.  ordered (x:xs@(y:rest)) = x <= y && ordered xs
```

De aceea a trebuit să specificăm și ce se întâmplă pe lista vidă, și ce se întâmplă pe lista cu un singur element

O definiție alternativă pentru funcția `ordered`:

```
1.  ordered2 (x:xs@(y:rest)) = x <= y && ordered2 xs
2.  ordered2 _ = True
```

Se traduce prin „orice altceva”
Unde în definiția lui ordered se mai putea folosi?

Când se poate folosi pattern matching

- De fiecare dată **când se leagă variabile**
  - La definirea funcțiilor
  - La crearea de legări locale folosind `let` sau `where` (vom vedea)

- Pattern-urile **nu** se potrivesc și între ele

```
eq x x = True   -- dă eroare Conflicting definitions for `x'
eq _ _ = False
```

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii `.` și `$`
- Expresii condiționale
- Legare statică
- Evaluare lenesă

Operatorii . și $

- **Operatorul .** (punct) realizează **compunere de funcții**

```
myLast = head . reverse -- myLast [1..5] =
myMin = head . sort     -- myMin [2,4,1,2,3,6,2] =
myMax = myLast . sort   -- myMax [2,4,1,2,3,6,2] =
```

- **Operatorul $** (dolar) realizează **aplicație de funcție**

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
```

  - `f $ a = f a` este interesant pentru că `$` are o **prioritate foarte mică**, astfel încât ambele părți vor fi evaluate înainte să se realizeze aplicația de funcție → evităm astfel să folosim foarte multe paranteze
  - `$` este **asociativ la dreapta** și este util pentru a rescrie structuri de genul **f (g (h … x))**, nu poate suplini orice fel de paranteze (vezi (odd . fst) mai sus)

Operatorii . și $

- **Operatorul .** (punct) realizează **compunere de funcții**

```
myLast = head . reverse -- myLast [1..5] = 5
myMin = head . sort     -- myMin [2,4,1,2,3,6,2] = 1
myMax = myLast . sort   -- myMax [2,4,1,2,3,6,2] = 6
```

- **Operatorul $** (dolar) realizează **aplicație de funcție**

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
```

  - `f $ a = f a` este interesant pentru că `$` are o **prioritate foarte mică**, astfel încât ambele părți vor fi evaluate înainte să se realizeze aplicația de funcție → evităm astfel să folosim foarte multe paranteze
  - `$` este **asociativ la dreapta** și este util pentru a rescrie structuri de genul **f (g (h … x))**, nu poate suplini orice fel de paranteze (vezi (odd . fst) mai sus)

Operatorii . și $

- **Operatorul .** (punct) realizează **compunere de funcții**

```
myLast = head . reverse -- myLast [1..5] = 5
myMin = head . sort     -- myMin [2,4,1,2,3,6,2] = 1
myMax = myLast . sort   -- myMax [2,4,1,2,3,6,2] = 6
```

- **Operatorul $** (dolar) realizează **aplicație de funcție**

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
                              --[(1,2),(3,4),(5,6),(7,8)]
```

  - `f $ a = f a` este interesant pentru că `$` are o **prioritate foarte mică**, astfel încât ambele părți vor fi evaluate înainte să se realizeze aplicația de funcție → evităm astfel să folosim foarte multe paranteze
  - `$` este **asociativ la dreapta** și este util pentru a rescrie structuri de genul **f (g (h … x))**, nu poate suplini orice fel de paranteze (vezi (odd . fst) mai sus)

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii `.` și `$`
- Expresii condiționale
- Legare statică
- Evaluare lenesă

Condiționala `if`

```
if condiție then rezultatThen else rezultatElse
```

Exemple

```
if 2<3 then "all normal" else "what did just happen?"  -- "all normal"

uglySum l = if null l then 0
                      else head l + uglySum (tail l)
```

Observație

- Un cod Haskell elegant va folosi pattern matching sau gărzi (vom vedea) înainte de a folosi `if`

Condiționala **case**

```
case expresie of
     pattern1 -> rezultat1
     pattern2 -> rezultat2
     ...
     patternn -> rezultatn
```

Are sens atunci când nu putem folosi pattern matching sau gărzi, de exemplu aici când verificăm structura lui head matrix, fără să o putem verifica pe a lui matrix în loc

Exemplu

```
myTranspose matrix = case (head matrix) of
      [] -> []
      _ -> map head matrix : myTranspose (map tail matrix)
```

Gărzi

```
f parametri
   | condiție1 = rezultat1
   | condiție2 = rezultat2
   ...
   | condițien = rezultatn
  [| otherwise = rezultatn]
```

← opțional

Exemplu

```
allEqual a b c
        | a==b = b==c
        | otherwise = False
```

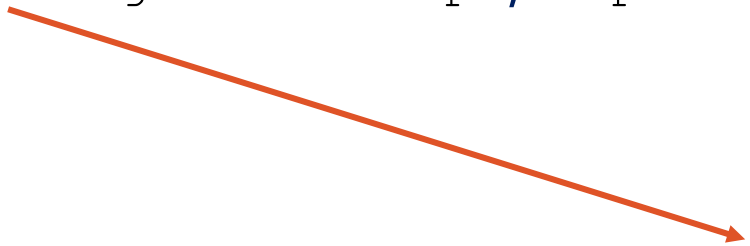
Au sens atunci când punem condiții asupra variabilelor, mai degrabă decât să le potrivim cu o anumită structură (caz în care am folosi pattern matching)

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii `.` și `$`
- Expresii condiționale
- Legare statică
- Evaluare lenesă

Legarea variabilelor în Haskell – statică

- Doar legare **statică**
- Expresii pentru legare locală: `let legări in expr`, `expr where legări`



Au sens pentru a spori lizibilitatea codului sau pentru a evita apelarea repetată a aceleiași funcții pe aceleași argumente

Exemple

```
myFoldl f acc [] = acc
myFoldl f acc (x:xs) =
        let
                newAcc = f acc x
        in myFoldl f newAcc xs

myFoldr f acc [] = acc
myFoldr f acc (x:xs) = f x rightResult
        where rightResult = myFoldr f acc xs
```

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii `.` și `$`
- Expresii condiționale
- Legare statică
- Evaluare lenesă

Evaluare lenesă

- Toate funcțiile sunt **nestricte**
- **Evaluare lenesă**: **subexpresiile (argumentele) sunt pasate funcției fără a fi evaluate, în corpul funcției ele se vor evalua (eventual parțial) maxim o dată**

Exemple

```
1. f x = 2*x
2. g x = f 2 + f 2
3. h x = x*x*x

g 5                 -- 2 aplicări distincte ale lui f => se evaluează de 2 ori
                    -- f 2 + f 2 -> 4 + f 2 -> 4 + 4 -> 8
h (f 2)             -- argumentul se evaluează o dată și se folosește de 3 ori
                    -- (f 2)*(f 2)*(f 2) -> 4*4*4 -> 64
```

Fluxuri

- Evaluare leneșă => toate **listele sunt fluxuri** (se evaluează în măsura în care e nevoie)

Exemple

```
naturals = let loop n = n : loop (n+1) in loop 0

ones = 1 : ones

fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)

evens = filter even naturals
```

Test

Definiți următoarele fluxuri în Racket:

- **`1, 1/2, 1/3, 1/4 ...`** – folosind o definiție explicită
- **`1, 1/2, 1/6, 1/12, 1/20 ...`** – folosind o definiție implicită

  (obs: `2=1*2, 6=2*3, 12=3*4` ...)

Generarea intervalelor

```
[start..stop] sau [start..]

[start,next..stop] sau [start,next..] --next dă pasul
```

Exemple

```
[1..5]

[1,3..10]

[10,7..0]

[20,19.5..]
```

Generarea intervalelor

```
[start..stop] sau [start..]
[start,next..stop] sau [start,next..] --next dă pasul
```

Exemple

```
[1..5]        -- [1,2,3,4,5]
[1,3..10]     -- [1,3,5,7,9]
[10,7..0]     -- [10,7,4,1]
[20,19.5..]   -- lista infinită [20,19.5,19,18.5..]
```

List comprehensions

```
[ expr | generatori, condiții, legări locale ]
```

Întâi toate rezultatele pentru x=1, apoi toate pentru x=2, apoi toate pentru x=3

Exemple

```
lc1 = [ (x,y,z) | x<-[1..3], y<-[1..4], x<y, let z = x+y, odd z ]

fibo = 0 : 1 : [ x+y | (x,y) <- zip fibo (tail fibo) ]

qsort [] = []
qsort (x:xs) =
      qsort [ y | y<-xs, y<=x ] ++
      [x] ++
      qsort [ y | y<-xs, y>x ]
```

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

Tipare tare / slabă

- **Tipare tare:** **nu se permit operații pe argumente care nu au tipul corect (se convertește tipul numai în cazul în care nu se pierde informație la conversie)**

  Exemplu: 1+"23" → eroare (Racket, Haskell)

- **Tipare slabă:** **nu se verifică corectitudinea tipurilor, se face cast după reguli specifice limbajului**

  Exemplu: 1+"23" = 24 (Visual Basic)
  1+"23" = "123" (JavaScript)

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

Tipare statică / dinamică

- **Tipare statică:** **verificarea tipurilor se face la compilare**
  – atât variabilele cât și valorile au un tip asociat

  Exemple: C++, Java, Haskell, ML, Scala, etc.

- **Tipare dinamică:** **verificarea tipurilor se face la execuție**
  – numai valorile au un tip asociat

  Exemple: Python, Racket, Prolog, Javascript, etc.

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

Tipuri primitive în Haskell

Tip

Bool = [True, False]

Char = [.. 'a', 'b', ..]

Int = [.. -1, 0, 1, ..]

Altele: **Integer**, **Float**, **Double**, etc.

Tipare expresie (:t expr)

True :: Bool

'a' :: Char

(fib 0) :: Int

Constructori de tip

Constructor de tip = „funcție” care creează un tip compus pe baza unor tipuri mai simple

- **(, , ...)** : $MT^n \longrightarrow MT$ (MT = mulțimea tipurilor)
  - $(t_1, t_2, \ldots, t_n)$ = **tuplu** cu elemente de tipurile $t_1, t_2, \ldots, t_n$
  - **Ex:** (Bool, Char) echivalent cu (,) Bool Char
- **[ ]** : $MT \longrightarrow MT$
  - [t] = **listă** cu elemente de tip t
  - **Ex:** [Int] echivalent cu [ ] Int
- **->** : $MT^2 \longrightarrow MT$
  - t_1 -> t_2 = **funcție** de un parametru de tip t_1 care calculează valori de tip t_2
  - **Ex:** Int -> Int echivalent cu (->) Int Int

Tipul funcțiilor n-are

Exemplu: `add x y = x + y` (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui `(add 2)`?

Tipul funcțiilor n-are

Exemplu: `add x y = x + y` (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui `(add 2)`?

```
(add 2) :: Int -> Int
```

- În aceste condiții, care este tipul lui `add`?

Tipul funcțiilor n-are

Exemplu: `add x y = x + y` (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui `(add 2)`?

```
(add 2) :: Int -> Int
```

- În aceste condiții, care este tipul lui `add`?

```
add :: Int -> (Int -> Int)
```

echivalent cu

```
add :: Int -> Int -> Int
```

întrucât **-> este asociativ la dreapta**

- Cum am interpreta tipul `(Int -> Int) -> Int`?

Tipul funcțiilor n-are

Exemplu: `add x y = x + y` (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui `(add 2)`?

```
(add 2) :: Int -> Int
```

- În aceste condiții, care este tipul lui `add`?

```
add :: Int -> (Int -> Int)
```

echivalent cu

```
add :: Int -> Int -> Int
```

întrucât **-> este asociativ la dreapta**

- Cum am interpreta tipul `(Int -> Int) -> Int`?

o funcție care – primește o funcție de la Int la Int
– întoarce un Int

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

Tipuri definite de utilizator

- Cuvântul cheie **data** dă utilizatorului posibilitatea definirii unui TDA cu implementare completă (constructori, operatori, axiome)

```
data NumeTip = Cons1 t11 .. t1i |
               Cons2 t21 .. t2j | ... |
               Consn tn1 .. tnk
```

Numele constructorilor valorilor tipului și tipurile parametrilor acestora (dacă au)

Exemple

```
data RH = Pos | Neg                    -- doar constructori nulari
data ABO = O | A | B | AB              -- doar constructori nulari
data BloodType = BloodType ABO RH      -- constructor extern
```

Exemplu – Tipul Natural

```
1.  data Natural = Zero | Succ Natural -- constructori nular și intern
2.                 deriving Show -- face posibilă afișarea valorilor tipului
3.  unu = Succ Zero
4.  doi = Succ unu
5.  trei = Succ doi
6.
7.  addN :: Natural -> Natural -> Natural -- arată exact ca axiomele
8.  addN Zero n = n
9.  addN (Succ m) n = Succ (addN m n)

addN unu trei                       -- Succ (Succ (Succ (Succ Zero)))
```

Constructorii valorilor unui TDA

Dublă utilizare a constructorilor valorilor unui TDA

- Compun noi valori pe baza celor existente (comportament de **funcție**)

  Exemple:
  ```
  unu = Succ Zero
  doi = Succ unu
  ```

- Descompun valori existente în scopul identificării structurii lor (comportament de **pattern**)

  Exemple:
  ```
  addN Zero n = n
  addN (Succ m) n = Succ (addN m n)
  ```

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

Tipuri parametrizate

- **Constructorii de valori** ale tipului (ex: **:**, **Succ**)
  - Pot primi valori ca argumente pentru a produce noi valori

    Exemple:
    ```
    unu = Succ Zero
    lista_unu = 1 : []
    ```

- **Constructorii de tip** (ex: **[ ]**, **(,)**, **->**)
  - Pot primi tipuri ca argumente pentru a produce noi tipuri

    Exemple:
    ```
    [Int], [Char], [[Char]]
    (Int, Char), ([Char], Int, Int)
    ```
  - Când TDA-ul sau funcțiile noastre se comportă la fel indiferent de tipul valorilor pe care le manipulează, folosim variabile (parametri) de tip

    Exemple:
    ```
    [a]     - o listă cu elemente de un tip oarecare a
    (a,b)   - o pereche de un element de un tip oarecare a
                        și un altul de un tip oarecare b
    ```

Exemplu – Tipul (Maybe a)

Tipul **(Maybe a)** există în Haskell și este definit astfel:

```
data Maybe a = Nothing | Just a
```

- Se folosește pentru situații când funcția întoarce sau nu un rezultat (de exemplu pentru funcții de căutare care ar putea să găsească sau nu ceea ce caută)
- În funcție de ce tip de valoare va stoca acest tip de date atunci când are ce stoca, constructorul de tip va produce un `Maybe Int` sau un `Maybe Char`, etc.

Exemplu de instanțiere (Maybe a)

Să se găsească suma pară maximă dintre sumele elementelor listelor unei liste de liste, dacă există (ex: `findMaxEvenSum [[1,2,3,4,5],[2,2],[2,4]] = Just 6`).

```
1. --findMaxEvenSum :: [[Int]] -> Maybe Int
2. findMaxEvenSum [] = Nothing
3. findMaxEvenSum (l:ls)
4.     | even lsum = case findMaxEvenSum ls of
5.             Just s -> Just (max lsum s)
6.             _ -> Just lsum
7.     | otherwise = findMaxEvenSum ls
8.       where lsum = sumL l
```

Din cauză că sumL este declarat ca
sumL :: Int -> Int
funcția va întoarce un (Maybe Int)

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Tipuri definite de utilizator
- Tipuri parametrizate
- Expresii de tip

Expresii de tip

- Expresiile reprezintă valori / expresiile de tip reprezintă tipuri

  Exemple: `Char, Int -> Int -> Int, (Char, [Int])`

- **Declararea signaturii** unei funcții (opțională în Haskell) **f :: exprDeTip**
  = asociere între numele funcției și o expresie de tip, cu rol de:
  - **Documentare** (ce ar trebui sa facă funcția)
  - **Abstractizare** (surprinde cel mai general comportament al funcției, funcția percepută ca operator al unui anumit TDA sau al unei clase de TDA-uri)
  - **Verificare** (Haskell generează o eroare dacă intenția (declarația) nu se potrivește cu implementarea)

Exemplu - myMap

`myMap :: (a -> b) -> [a] -> [b]` trebuie să:

primească: o funcție de la un tip a la un tip b
o listă de elemente de tip a

întoarcă: o listă de elemente de tip b

- Dacă implementăm myMap astfel:

```
1. myMap :: (a -> b) -> [a] -> [b]
2. myMap f [] = []
3. myMap f (x:xs) = f (f x) : myMap f xs
```

compilatorul va da eroare, arătând că funcția nu se comportă conform declarației.

- Fără declarația de tip, Haskell ar fi dedus singur tipul lui myMap și ne-ar fi lăsat să continuăm cu o funcție care nu face ceea ce dorim.

Observații

Verificarea strictă a tipurilor înseamnă:

- Mai **multă siguranță** („dacă trece de compilare atunci merge”)
- Mai **puțină libertate**
  - Listele sunt neapărat omogene: **[a]**
    - Contrast cu liste ca `'(1 'a #t)` din Racket
  - Funcțiile întorc mereu valori de un același tip: **f :: ... -> b**
    - Contrast cu funcții ca **`member`** din Racket (care întoarce o listă sau #f)

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional		
Funcții		
Pattern matching		
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții		
Pattern matching		
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching		
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare	Locală – statică, top-level - dinamică	Statică
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare	Locală – statică, top-level - dinamică	Statică
Evaluare	Aplicativă	Lenesă
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare	Locală – statică, top-level - dinamică	Statică
Evaluare	Aplicativă	Lenesă
Tipare	Tare, dinamică	Tare, statică

Rezumat

Perechi și liste
Funcții
. și $
Expresii condiționale
Expresii pentru legare locală
Evaluare lenește
List comprehensions
Tipare tare/slabă
Tipare statică/dinamică
Constructori de tip
Tipuri definite de utilizator
Tipuri parametrizate
Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții

. și $

Expresii condiționale

Expresii pentru legare locală

Evaluare lenesă

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $

Expresii condiționale

Expresii pentru legare locală

Evaluare lenește

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale

Expresii pentru legare locală

Evaluare lenesă

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală

Evaluare lenesă

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Tipuri definite de utilizator

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Tipuri definite de utilizator: data NumeTip = $Cons_1\ t_{11} .. t_{1i} \mid ... \mid Cons_n\ t_{n1} .. t_{nk}$

Tipuri parametrizate

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Tipuri definite de utilizator: data NumeTip = $Cons_1\ t_{11} .. t_{1i} \mid ... \mid Cons_n\ t_{n1} .. t_{nk}$

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Declararea signaturii

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și $: compunere de funcții / aplicație de funcție

Expresii condiționale: if, case, gărzi

Expresii pentru legare locală: let, where

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Tipuri definite de utilizator: data NumeTip = $Cons_1\ t_{11} .. t_{1i} \mid ... \mid Cons_n\ t_{n1} .. t_{nk}$

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Declararea signaturii: f :: exprTip