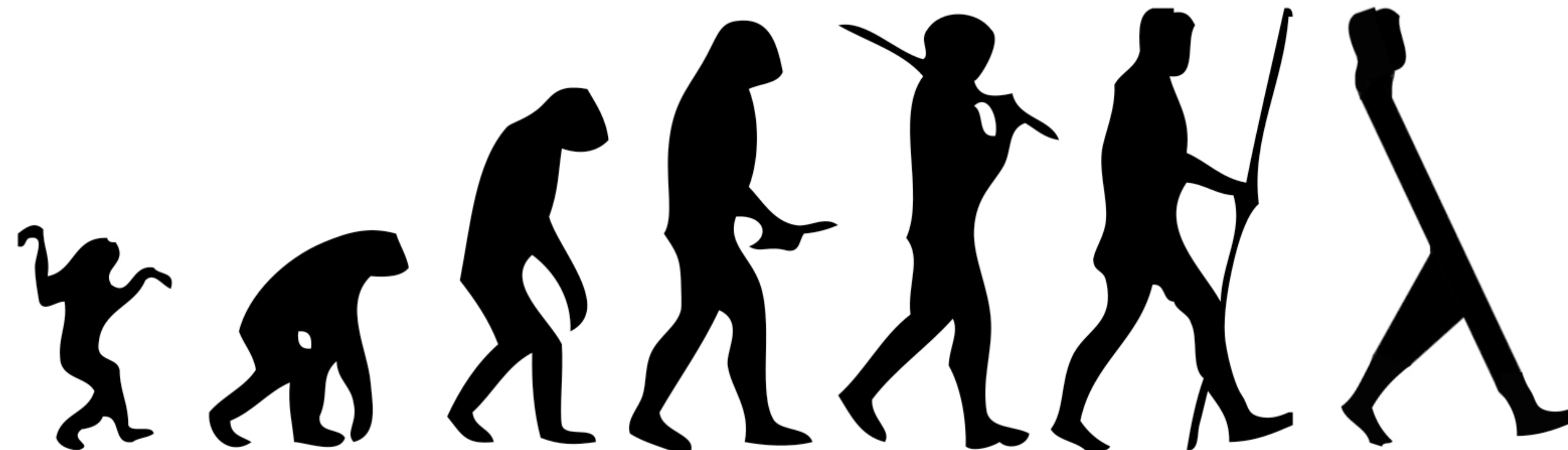
PARADIGME DE PROGRAMARE

Curs 12

Limbajul Haskell.

Programare funcțională în Haskell



Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Expresii condiționale
- Funcționale
- Legare statică

Sintaxa Haskell

- Funcții și operatori
  - Operatori (infixat): `1 + 2, a < 5`
  - Funcții (prefixat): `filter odd [1 .. 7], f 2 + f 5`
- Paranteze
  - Controlează prioritatea (nu aplicația de funcție!): `f (1 + 2), f (g x)`
  - Reguli privind aplicația de funcție:
    - Prioritate maximă: `(f 1) + 2 ≡ f 1 + 2`
    - Asociativitate la stânga: `(f g) x ≡ f g x`
- Indentare
  - înlocuiește controlul prin separatori ca { } sau ;
  - Corpul unei expresii trebuie indentat la dreapta față de începutul expresiei
  - Expresie nouă: începe pe același nivel sau la stânga față de începutul expresiei anterioare

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Expresii condiționale
- Funcționale
- Legare statică

TDA-ul Pereche

Constructori

(,) :: a -> b -> (a, b) // creează o pereche între orice 2 argumente

Operatori

fst :: (a, b) -> a // extrage prima valoare din pereche
snd :: (a, b) -> b // extrage a doua valoare din pereche

Exemple

```
(1, "unu")

a = (('a', 2), "a2")

fst a

snd (fst a)
```

TDA-ul Pereche

Constructori

(,) :: a -> b -> (a, b) // creează o pereche între orice 2 argumente

Operatori

fst :: (a, b) -> a // extrage prima valoare din pereche

snd :: (a, b) -> b // 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

[ ] :: [a] // creează o listă vidă

(:) :: a -> [a] -> [a] // creează o listă prin adăugarea unei valori la începutul unei liste

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

Operatori

head :: [a] -> a

tail :: [a] -> [a]

null :: Foldable t => t a -> Bool

length :: Foldable t => t a -> Int

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

Exemple

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

TDA-ul Listă

Constructori

[ ] :: [a]	// creează o listă vidă
(:) :: a -> [a] -> [a]	// creează o listă prin adăugarea unei valori la începutul unei liste
[, , .. ,] :: a x .. a -> [a]	// creează o listă din toate argumentele sale (de același tip)

Operatori

- **head** :: [a] -> a
- **tail** :: [a] -> [a]
- **null** :: Foldable t => t a -> Bool
- **length** :: Foldable t => t a -> Int
- **(++)** :: [a] -> [a] -> [a]

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
- Expresii condiționale
- Funcționale
- Legare statică

Funcții anonime în Haskell

```
\parametri -> corp
```

Exemple

$\lambda x.x$	`\x -> x`	
$\lambda x.\lambda y.(x\ y)$	`\x -> \y -> x y`	$\equiv$
	`\x y -> x y`	(funcțiile Haskell sunt automat curry)
$(\lambda x.x\ \lambda x.y)$	`(\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
```

Simularea funcțiilor uncurry

Funcții curry (standard): $f\ x_1\ x_2\ \ldots\ x_n = corp$

- Aplicația $f\ e_1\ e_2 \ldots e_k$ pe $k < n$ argumente întoarce o nouă funcție $\backslash x_{k+1}\ \ldots\ x_n \to corp_{[ei/xi]}$.

Funcții uncurry (cu parametru de tip tuplu): $f\ (x_1,\ x_2\ \ldots,\ x_n) = corp$

- ex: `arithmeticMean (x, y) = (x + y)/2`
- apel valid: `arithmeticMean (3,18)`
- apel invalid: `arithmeticMean 3` (produce o eroare de tip)

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, -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
- Expresii condiționale
- Funcționale
- Legare statică

Definirea funcțiilor prin pattern matching

- 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
```

permite crearea unui alias pentru valoarea următoare

La aplicare – ordinea contează!

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

fib 3
```

- Dacă argumentul se potrivește cu primul pattern
  - Folosește definiția din primul punct
  - Ignoră definițiile următoare
- Altfel
  - Încearcă potrivirea cu punctul următor, etc.

Şabloane exhaustive

Şabloanele trebuie să acopere toate valorile tipului, prevenind erori la execuție.

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

Trebuie specificat și ce se întâmplă pe lista vidă, și ce se întâmplă pe lista cu un singur element

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?

Pattern matching – utilizare

- De fiecare dată când se leagă variabile
  - La definirea funcțiilor – legarea se efectuează la apelul pe argumente
  - La crearea de legări locale – folosind `let` sau `where` (vom vedea)

- Șabloanele nu se potrivesc între ele

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

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Expresii condiționale
- Funcționale
- Legare statică

Condiționala `if`

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

Exemplu

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

Observații

- Ambele ramuri (`then`, `else`) sunt obligatorii!
- `rezultatThen` și `rezultatElse` trebuie să aibă același tip!
- `if` este considerat neelegant, pe lângă pattern matching, gărzi sau `case` (vom vedea)

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)

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, mai degrabă decât pe a lui matrix

Exemplu

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

Gărzi cu potrivire (pattern guards)

```
f parametri
  | garda₁, garda₂, ...  = rezultat₁
  ...
  | gardaₙ = rezultatₙ
 [| otherwise = rezultatₙ]
```

Extensie a gărzilor standard, combină testarea condițiilor cu legarea locală sau legarea prin pattern matching

Garda poate fi:

- expresie booleană `x > 0`
- legare cu pattern matching `x:xs <- l`
- legare locală `let odds = filter odd xs`

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Expresii condiționale
- Funcționale
- Legare statică

Funcționale

Funcționale

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

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

foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b

foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

Exemple

```
map (:[]) [2,3]
filter id [True, False]
foldl min 6 [1..5]
foldr (:) [] [1..3]
zipWith (+) [1..3] [1..4]
```

Observații:

- Funcția binară primită ca parametru de foldl are acumulatorul pe stânga!
- map, foldl și foldr au signaturi bine precizate și nu pot prelucra mai multe liste în paralel (există însă zipWith, zipWith3, zipWith4, etc.)

Funcționale

Funcționale

map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

Exemple

```
map (:[]) [2,3]     -- [[2],[3]]
filter id [True, False]   -- [True]
foldl min 6 [1..5]        -- 1
foldr (:) [] [1..3]       -- [1,2,3]
zipWith (+) [1..3] [1..4] -- [2,4,6]
```

Observații:

- Funcția binară primită ca parametru de foldl are acumulatorul pe stânga!
- map, foldl și foldr au signaturi bine precizate și nu pot prelucra mai multe liste în paralel (există însă zipWith, zipWith3, zipWith4, etc.)

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Expresii condiționale
- Funcționale
- Legare statică

Legarea variabilelor

- Haskell – doar legare statică
- Expresii pentru legare locală:
  - `let legări in expr`
  - `expr where legări`

→

- cresc lizibilitatea codului
- evită apelarea repetată a aceleiași funcții pe aceleași argumente

Exemplu

```
1.   solveQuad a b c =
2.     let delta = b^2 - 4*a*c
3.         x1 = (-b + sqrt delta) / (2*a)
4.         x2 = (-b - sqrt delta) / (2*a)
5.     in (x1, x2)
6.       where sqrt = (**0.5)
```

Rezumat

Perechi

Liste

Funcții

Condiționale

Funcționale

Legare locală

Rezumat

Perechi: Constructor: (1,'a') Operatori: fst, snd

Liste

Funcții

Condiționale

Funcționale

Legare locală

Rezumat

Perechi: Constructor: (1,'a') Operatori: fst, snd

Liste: Constructori: [1,2,3], [], (:) Operatori: null, head, tail, length, (++)

Funcții

Condiționale

Funcționale

Legare locală

Rezumat

Perechi: Constructor: (1,'a') Operatori: fst, snd

Liste: Constructori: [1,2,3], [], (:) Operatori: null, head, tail, length, (++)

Funcții: \x y -> corp, f x y = corp

Condiționale

Funcționale

Legare locală

Rezumat

Perechi: Constructor: (1,'a') Operatori: fst, snd

Liste: Constructori: [1,2,3], [], (:) Operatori: null, head, tail, length, (++)

Funcții: \x y -> corp, f x y = corp

Condiționale:

```
if cond then rez1 else rez2
case expr of
    pattern1 -> rez1 ...

f ...
    | garda1, garda2, ... = rez1   ...
```

Funcționale

Legare locală

Rezumat

Perechi: Constructor: (1,'a') Operatori: fst, snd

Liste: Constructori: [1,2,3], [], (:) Operatori: null, head, tail, length, (++)

Funcții: \x y -> corp, f x y = corp

Condiționale:

```
if cond then rez1 else rez2
case expr of
    pattern1 -> rez1 ...

f ...
    | garda1, garda2, ... = rez1   ...
```

Funcționale: map, filter, foldl, foldr, zipWith

Legare locală

Rezumat

Perechi: Constructor: (1,'a') Operatori: fst, snd

Liste: Constructori: [1,2,3], [], (:) Operatori: null, head, tail, length, (++)

Funcții: \x y -> corp, f x y = corp

Condiționale:

```
if cond then rez1 else rez2
case expr of
    pattern1 -> rez1 ...

f ...
    | garda1, garda2, ... = rez1   ...
```

Funcționale: map, filter, foldl, foldr, zipWith

Legare locală:

```
let legări in expr
expr where legări
```