

Paradigme de Programare

Conf. dr. ing. Andrei Olaru

andrei.olaru@upb.ro

Departamentul de Calculatoare

2024

Cursul 6: Programare funcțională în Haskell



```
primes = sieve [2..]
  where sieve (p:xs) =
    p : sieve [x | x <- xs, x `mod` p /= 0]
```



- 1 Introducere
- 2 Sintaxă
- 3 Evaluare



Alt text: The problem with Haskell is that it's a language built on lazy evaluation and nobody's actually called for it.

Introducere



[[https://en.wikipedia.org/wiki/Haskell_\(programming_language\)](https://en.wikipedia.org/wiki/Haskell_(programming_language))]

- din 1990;
- GHC – Glasgow Haskell Compiler (The Glorious Glasgow Haskell Compilation System)
 - dialect Haskell standard *de facto*;
 - compilează în/folosind C;
- Haskell Stack
- nume dat după logicianul Haskell Curry;
- aplicații: Pugs, Darcs, Linspire, Xmonad, Cryptol, seL4, Pandoc, web frameworks.



Criteriu	Racket	Haskell
Funcții	<i>Curry</i> sau <i>uncurry</i>	<i>Curry</i>
Tipare	Dinamică, tare (-liste)	Statică, tare
Legarea variabilelor	Statică	Statică
Evaluare	Aplicativă	Normală (Leneșă)
Transferul parametrilor	<i>Call by sharing</i>	<i>Call by need</i>
Efecte laterale	set!*	Interzise

Sintaxă



- toate funcțiile sunt *Curry*;
- aplicabile asupra **oricâtor** parametri la un moment dat.

Ex | Exemplu : Definiții **echivalente** ale funcției `add`:

- toate funcțiile sunt *Curry*;
- aplicabile asupra **oricâtor** parametri la un moment dat.

Ex | Exemplu : Definiții **echivalente** ale funcției `add`:

```
1 add1           = \x y -> x + y
2 add2           = \x -> \y -> x + y
3 add3 x y       = x + y
4
5 result         = add1 1 2      -- echivalent, ((add1 1) 2)
6 result2        = add3 1 2      -- echivalent, ((add3 1) 2)
7 inc            = add1 1
```



- Aplicabilitatea **parțială** a operatorilor infixati
- **Transformări** operator \rightarrow funcție și funcție \rightarrow operator



- Aplicabilitatea **parțială** a operatorilor infixati
- **Transformări** operator \rightarrow funcție și funcție \rightarrow operator

Ex Exemplu Definiții **echivalente** ale funcțiilor `add` și `inc`:

```
1 add4           = (+)
2 result1       = (+) 1 2
3 result2       = 1 'add4' 2
4
5 inc1          = (1 +)
6 inc2          = (+ 1)
7 inc3          = (1 'add4')
8 inc4          = ('add4' 1)
```

- Definirea comportamentului funcțiilor pornind de la **structura** parametrilor
→ traducerea axiomelor TDA.

Ex Exemplu

```
1 add5 0 y          = y          -- add5 1 2
2 add5 (x + 1) y   = 1 + add5 x y
3
4 sumList []        = 0          -- sumList [1,2,3]
5 sumList (hd:tl)   = hd + sumList tl
6
7 sumPair (x, y)    = x + y      -- sumPair (1,2)
8
9 sumTriplet (x, y, z@(hd:_)) = -- sumTriplet
10    x + y + hd + sumList z      -- (1,2,[3,4,5])
```



- Definirea listelor prin **proprietățile** elementelor, ca într-o specificare matematică

Ex Exemplu

```
1 squares lst      = [x * x | x <- lst]
2
3 quickSort []     = []
4 quickSort (h:t) = quickSort [x | x <- t, x <= h]
5                 ++ [h]
6                 ++ quickSort [x | x <- t, x > h]
7
8 interval         = [0 .. 10]
9 evenInterval     = [0, 2 .. 10]
10 naturals        = [0 ..]
```

Evaluare



- Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcții **nestricte!**

Ex Exemplu

```
1 f (x, y) z = x + x
```

Evaluare:

```
1 f (2 + 3, 3 + 5) (5 + 8)
```

```
2
```

```
3
```

```
4
```




- Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcții **nestricte**!

Ex Exemplu

```
1 f (x, y) z = x + x
```

Evaluare:

```
1 f (2 + 3, 3 + 5) (5 + 8)
```

```
2 → (2 + 3) + (2 + 3)
```

```
3
```

```
4
```



- Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcții **nestricte**!

Ex Exemplu

1 $f(x, y) z = x + x$

Evaluare:

1 $f(2 + 3, 3 + 5) (5 + 8)$

2 $\rightarrow (2 + 3) + (2 + 3)$

3 $\rightarrow \underline{5} + \underline{5}$ **reutilizăm** rezultatul primei evaluări!

4 $\rightarrow 10$ ceilalți parametri **nu sunt evaluați**

Pași în aplicarea funcțiilor

Exemplu



Ex Exemplu

```
1 frontSum (x:y:zs) = x + y
2 frontSum [x]      = x
3
4 notNil []         = False
5 notNil (_:_)     = True
6
7 frontInterval m n
8   | notNil xs = frontSum xs
9   | otherwise = n
10  where
11    xs = [m .. n]
```



- 1 **Pattern matching**: evaluarea parametrilor **suficient** cât să se constate (ne-)potrivirea cu *pattern*-ul;
- 2 Evaluarea **gărzilor** (|);
- 3 Evaluarea variabilelor **locale**, **la cerere** (`where`, `let`).

Pași în aplicarea funcțiilor

Exemplu – revisited



Ex execuția exemplului anterior

```
1 frontInterval 3 5
2 ?? notNil xs
3 ?? where
4 ?? xs = [3 .. 5]
5 ?? → 3:[4 .. 5]
6 ?? → notNil (3:[4 .. 5])
7 ?? → True
8 → frontSum xs
9 where
10 xs = 3:[4 .. 5]
11 → 3:4:[5]
12 → frontSum (3:4:[5])
13 → 3 + 4 → 7
```

evaluare pattern

evaluare prima gardă

necesar `xs` → evaluare where

evaluare valoare gardă

`xs` deja calculat

- Evaluarea **parțială** a structurilor – liste, tupluri etc.
- Listele sunt, implicit, văzute ca **fluxuri**!

Ex Exemplu

```
1 ones           = 1 : ones
2
3 naturalsFrom n = n : (naturalsFrom (n + 1))
4 naturals1      = naturalsFrom 0
5 naturals2      = 0 : (zipWith (+) ones naturals2)
6
7 evenNaturals1  = filter even naturals1
8 evenNaturals2 = zipWith (+) naturals1 naturals2
9
10 fibo          = 0 : 1 : (zipWith (+) fibo (tail fibo))
```



- Haskell, diferențe față de Racket
- pattern matching și list comprehensions
- evaluare în Haskell

+ Dați feedback la acest curs aici:

[<https://docs.google.com/forms/d/e/1FAIpQLSeY7VuAt5n6hyHHnNUp1LWfWt7UkJBGhkviewform>]

