Programare declarativă[1]

Introducere în programarea funcțională folosind Haskell

Traian Florin Șerbănuță
Ioana Leuștean

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

[1] bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Sintaxă

- Comentarii

```
-- comentariu pe o linie
{- comentariu pe
   mai multe
   linii -}
```

- Identificatori
  - siruri formate din litere, cifre, caracterele _ si ’ (single quote)
  - incep cu o litera
- Haskell este case sensitive
  - identificatorii pentru variabile incep cu litera mica
  - identificatorii pentru constructori incep cu litera mare

```
let double x = 2 * x
data Point a = Pt a a
```

Sinatxa

Blocuri si indentare

Blocurile sunt delimitate prin indentare.

```
fact n =  if n == 0 then 1
          else  n * fact(n-1)

trei =  let
            a = 1
            b = 2
        in (a + b)
```

- echivalent, putem scrie

```
trei  =  let a = 1; b = 2 in (a + b)
```

Module

Program in Haskell

Un program in Haskell este o colectie de module.

- modulele contin declaratii de functii, tipuri si clase
- modulele sunt scrise in fisiere; un fisier contine un singur modul, numele fisierului coincide cu numele modulului si incepe cu litera mare

```
module MyDouble where

double :: Integer -> Integer
double x = x + x
```

- modulele pot fi importate

```
import MyDouble
```

Variabile

= reprezinta o legatura (binding)

In Haskell, variabilele sunt imuabile

- daca fisierul test.hs contine

  x=1
  x=2

```
Prelude> :l test.hs

test.hs:2:1: error:
    Multiple declarations of ‘x’
    Declared at: test.hs:1:1
                 test.hs:2:1
  |
2 | x=2
  | ^
```

Legarea variabilelor

let .. in ...

creaza scop local

- daca fisierul testlet.hs contine

```
x=1
z= let x=3 in x
```

```
Prelude> :l testlet.hs
[1 of 1] Compiling Main
Ok, 1 module loaded.
*Main> z
3
*Main> x
1
```

Legarea variabilelor

- **let .. in ...** creaza scop local

```
x = let
        z = 5
        g u = z + u
    in  let
            z = 7
        in (g 0 + z)     -- x=12
```

Legarea variabilelor

- **let .. in ...** creaza scop local

```
x = let
        z = 5
        g u = z + u
    in  let
            z = 7
        in (g 0 + z)    -- x=12
```

```
x= let z=5; g u = z+u in let z=7 in g 0 -- x=5
```

Legarea variabilelor

- **let .. in ...** creaza scop local

```
x = let
        z = 5
        g u = z + u
    in  let
            z = 7
        in (g 0 + z)    -- x=12
```

```
x= let z=5; g u = z+u in let z=7 in g 0 -- x=5
```

- **... where ...** creaza scop local

```
f x = (g x) + (g x) + z
      where g x = 2*x
            z =  x-1
```

Legarea variabilelor

- **let .. in ...** este o expresie

```
x = [let y =8 in y, 9] -- x=[8,9]
```

- **where** este o clauza

```
x = [y where y =8, 9] -- error: parse error ...
```

Clauza **where** poate fi folosita pentrua defini functii si expresii **case**.

```
h x | x == 0 = 0
    | x == 1 = y + 1
    | x == 2 = y * y
    | otherwise = y
   where y = x*x
```

```
f x = case x of
        0 -> 0
        1 -> y + 1
        2 -> y * y
        otherwise -> y
        where y = x*x
```

Expresii și funcții

Signatura unei funcții

```
fact :: Integer -> Integer
```

- Definiții folosind if

```
fact n = if n == 0 then 1
         else n * fact(n-1)
```

- Definiții folosind ecuații

```
fact 0 = 1
fact n = n * fact(n-1)
```

- Definiții folosind cazuri

```
fact n
  | n == 0     = 1
  | otherwise = n * fact(n-1)
```

Șabloane (patterns)

- `x:y = [1,2,3]` -- *x=1 si y =[2,3]*

  Observati ca : este constructorul pentru liste

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

  `(_,_)` este un tip compus

- Definitii folosind case...of

```
selectie :: (Integer, String) -> String

selectie (x,s) = case (x,s) of
                      (0,_) -> s
                      (1, z:zs) -> zs
                      (1, []) -> []
                      (_,_) -> (s ++ s)
```

Definiții de liste

- 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]
progresieInfinita = [3,7..]    -- [3,7,11,15,19,..]
```

- Definiții prin selecție

```
pare :: [Integer] -> [Integer]
pare xs = [x | x<-xs, even x]

pozitiiPare :: [Integer] -> [Integer]
pozitiiPare xs = [i | (i,x) <- [1..] `zip` xs, even x]
```

Sistemul tipurilor

"There are three interesting aspects to types in Haskell: they are strong, they are static, and they can be automatically inferred."
http://book.realworldhaskell.org/read/types-and-functions.html

tare garanteaza absenta anumitor erori

static tipul fiecari valori este calculat la compilare

dedus automat compilatorul deduce automat tipul fiecarei expresii

```
Prelude> :t [('a',1,"abc")]
[('a',1,"abc")] :: Num b => [(Char, b, [Char])]
```

Sistemul tipurilor

Tipurile de baza

Int Integer Float Double Bool Char String

- tipuri compuse: tupluri si liste

```
Prelude> :t [('a',1,"abc")]
[('a',1,"abc")] :: Num b => [(Char, b, [Char])]
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
```

Tipuri. Clase de tipuri. Variabile de tip. Signaturi de tip

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t "ana"
"ana" :: [Char]
Prelude> :t 1
1 :: Num a => a
Prelude> :t [1,2,3]
[1,2,3] :: Num t => [t]
Prelude> :t 3.5
3.5 :: Fractional a => a
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (+3)
(+3) :: Num a => a -> a
Prelude> :t (3+)
(3+) :: Num a => a -> a
```

Expresii ca valori

Funcțiile — „cetățeni de rangul I”

- Funcțiile sunt valori care pot fi luate ca argument sau întoarse ca rezultat

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \ x y -> f y x
-- sau alternativ folosind matching
flip f x y = f y x
-- sau flip ca valoare de tip functie
flip = \ f x y -> f y x
-- Currying
flip = \f -> \x -> \y -> f y x
```

- Aplicare parțială a funcțiilor

```
injumatateste :: Integral a => a -> a
injumatateste = (`div` 2)
```

Funcții de ordin înalt

map, filter, foldl, foldr

```
Prelude> map (*3) [1,3,4]
[3,9,12]
Prelude> filter (>=2) [1,3,4]
[3,4]
Prelude> foldr (*) 1  [1,3,4]
12
Prelude> foldl (flip (:)) [] [1,3,4]
[4,3,1]
```

Compunere si aplicare

```
Prelude> map (*3) ( filter (<=3) [1,3,4])
[3,9]
Prelude> map (*3) . filter (<=3)  [1,3,4]
[3,9]
```

Lenevire

- Argumentele sunt evaluate doar cand e necesar si doar cat e necesar

```
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> let x = head []
Prelude> let f a = 5
Prelude> f x
5
Prelude> head [1,head [],3]
1
Prelude> head [head [],3]
*** Exception: Prelude.head: empty list
```

- Liste infinite (fluxuri de date)

```
ones = [1,1..]
zeros = [0,0..]
both = zip ones zeros
short = take 5 both   -- [(1,0),(1,0),(1,0),(1,0),(1,0)]
```

Intteractiune cu mediul extern

- Monade
- Acțiuni
- Secvențiere