IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Haskell

Notiuni de baza

Ioana Leustean

- **Comentarii**

```
-- comentariu pe o linie

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

- **Identificatorii**

- 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
```

let si data apartin limbajului

- **Blocurile sunt delimitate prin indentare**
  se pot folosi ; si { } dar nu e uzual
  evitati folosirea tab-urilor

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

suma  =  let
              a = 1
              b = 2
              c = 3
           in (a + b + c)
```

if .. then .. else si let.. in sunt expresii

```
main =
    do
         print "What is your name?"
         name <- getLine
         print ("Hello " ++ name )
```

```
*Main> :t main
main :: IO ()
*Main> :t getLine
getLine :: IO String
*Main> :t print
print :: Show a => a -> IO ()
```

Intrarile si iesirile sunt reprezentate prin actiuni care sunt valori de tipul IO

```
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :cd D:\DIR\HS\myexc
Prelude> :l myfact.hs
[1 of 1] Compiling Main             ( myfact.hs, interpreted )
Ok, modules loaded: Main.
*Main> fact 7
5040
*Main>
```

myfact.hs

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

```
Prelude> :l myhello.hs
[1 of 1] Compiling Main             ( myhello.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
"What is your name?"
Ioana
"Hello Ioana"
```

myhello.hs

```
main =
    do
        print "What is your name?"
        name <- getLine
        print ("Hello " ++ name )
```

- **Program in Haskell:**
  - colectie de module, care pot fi importate;
  - 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.

Mymod.hs

```
module Mymod where

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

```
Prelude> :l Mymod.hs
[1 of 1] Compiling Mymod            ( Mymod.hs, interpreted )
Ok, modules loaded: Mymod.
*Mymod> double 7
14
*Mymod> :m
Prelude> :l import-mod.hs
[1 of 2] Compiling Mymod            ( Mymod.hs, interpreted )
[2 of 2] Compiling Main             ( import-mod.hs, interpreted )
Ok, modules loaded: Main, Mymod.
*Main> main
"A number"
5
10
*Main>
```

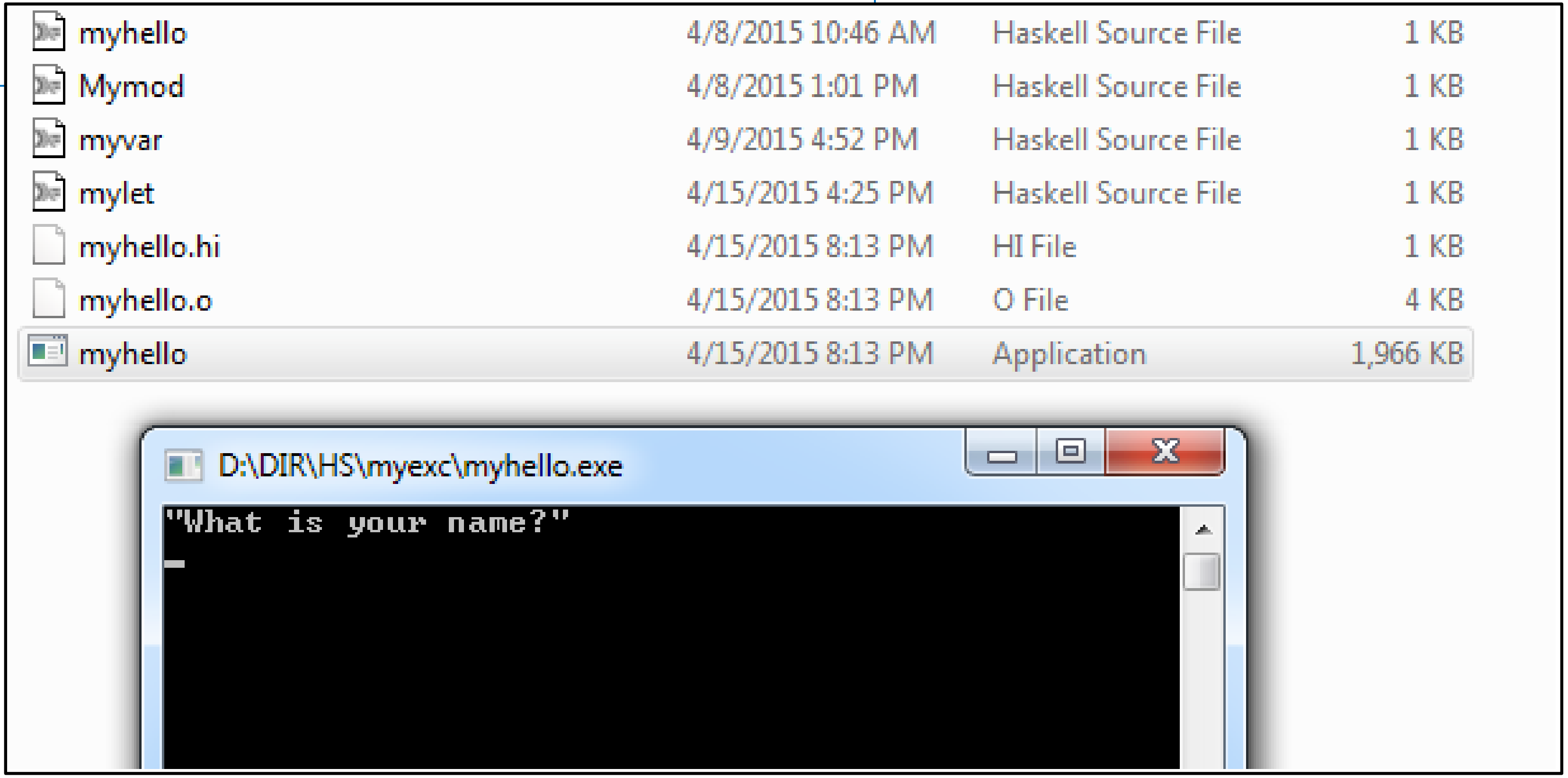
import-mod.hs

```
import Mymod

main =
  do
     print "A number"
     nr <- readLn
     print ( double nr)
```

➢ Programele pot fi compilate

```
Prelude>  :! ghc --make "myhello"
[1 of 1] Compiling Main             ( myhello.hs, myhello.o )
Linking myhello.exe ...
Prelude>
```



A rula un program Haskell compilat revine la a executa functia main

Valoarea functiei main este o **actiune de tipul** IO ()

```
Prelude> :l myhello.hs
Ok, modules loaded: Main.
Prelude Main> :t main
main :: IO ()
Prelude Main>
```

➢ **Variabile imuabile ("immutable")**

x = 7 "x este 7"

= "binding"

myvar.hs

```
x = y + 3
y = 7
```

```
Prelude> :l myvar.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> x
10
```

let .. in creaza scop

```
Prelude> let x = 7 in let x = 3 in x
3
```

myvar.hs

```
x = y + 3
y = 7

x = 4
-- x = x + 1
```

```
Prelude> :l myvar.hs

myvar.hs:4:1:
    Multiple declarations of ‘x’
    Declared at: myvar.hs:1:1
                 myvar.hs:4:1
[1 of 1] Compiling Main
Failed, modules loaded: none.
```

- **Expresia let … in**

let … in creaza scop local

```
x = let
      z = 5
      g u = z + u
    in  g 0

-- x = 5
```

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

-- x = 5
```

```
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  g 0
```

- Expresia let ... in
- Clauza where

sunt constructiile care leaga variabilele local

```
Prelude> let  x = let a =1; b=2; c=3 in (a + b + c)
Prelude> x
6
Prelude> let x = a + b + c where a=1; b=2; c=3
Prelude> x
6
```

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

```
f:: Int -> Int
f x = let g x =  2* x
          z = x -1
      in  (g x) + (g x) + z
```

```
Prelude> let z = 5 in let g u = z + u in let z = 7 in g 0
5
```

- let poate fi folosit in 3 moduri

  - expresia let … in

  - declaratia let in blocul do (fara in)

  - declaratia let in definirea listelor prin comprehensiune (fara in)
    [(x, y) | x <- [1..3], let z= x*x; y = z-1]

- **Lazy evaluation (call-by-need)**
  expresiile sunt evaluate atunci cand valoarea lor este solicitata

```
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> let x = head []
Prelude> :sprint x
x = _
Prelude> let f y = 4
Prelude> f x
4
```

:sprint afiseaza valoarea unei expresii fara a forta evaluarea

_ este un symbol special care arata ca expresia nu este evaluata

```
Prelude>  True && (x == "a")
*** Exception: Prelude.head: empty list
Prelude>  False && (x == "a")
False
```

```
Prelude> let x = x + 1
Prelude> :sprint x
x = _
Prelude> x
*** Exception: <<loop>>
```

➢ Structuri de date infinite

```
Prelude> let inflist = 1 : inflist
Prelude> inflist
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

```
Prelude> let inflist = 1 : inflist
Prelude> let zece = take 10 inflist
Prelude> zece
[1,1,1,1,1,1,1,1,1,1]
```

```
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1Prelude>
Prelude> let zece = take 10 inflist
Prelude> zece
[1,1,1,1,1,1,1,1,1,1]
```

take este o functie definita in Prelude

Prelude> :t take
take :: Int -> [a] -> [a]

➢ Sistemul tipurilor

"There are three interesting aspects to types in Haskell:
they are *strong*, they are *static*, and they can be automatically *inferred*."
Real World Haskell

tare garanteaza absenta anumitor erori
static tipul fiecari valori este calculat la compilare
dedus automat compilatorul deduce automat tipul fiecarei expresii

```
Prelude> let list = ["lista", "de", "cuvinte"]
Prelude> :t list
list :: [[Char]]
Prelude>
```

```haskell
module Mymod where

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

```
*Mymod> double 5
10
*Mymod> double 4.5

<interactive>:93:8:
    No instance for (Fractional Integer) arising from the literal ‘4.5’
    In the first argument of ‘double’, namely ‘4.5’
    In the expression: double 4.5
    In an equation for ‘it’: it = double 4.5
*Mymod>
```

➢ [Tipuri de baza](#)

Int Integer Float Double

```
Prelude> maxInt
9223372036854775807
Prelude> let minInt = minBound :: Int
Prelude> minInt
-9223372036854775808
Prelude> let f :: Integer -> Integer; f n = product[1..n]
Prelude> f 30
265252859812191058636308480000000
```

```
Prelude> 2 + 3
5
Prelude> (+) 2 3
5
```

```
Prelude> let z = 2 * pi * 5 :: Float
Prelude> z
31.415928
Prelude> let y = 2 * pi * 5 :: Double
Prelude> y
31.41592653589793
```

```
Prelude> truncate y
31
Prelude> round y
31
Prelude> ceiling y
32
Prelude> floor y
31
```

100!

```
Prelude> f 100
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
```

➢ Tipuri de baza

Int Integer Float Double

```
Prelude> sqrt 4.5
2.1213203435596424
Prelude> sqrt 4
2.0
Prelude> let x = 4 :: Integer
Prelude> x
4
Prelude> sqrt x

<interactive>:248:1:
    No instance for (Floating Integer) arising from a use of ‘sqrt’
    In the expression: sqrt x
    In an equation for ‘it’: it = sqrt x
Prelude> sqrt (fromIntegral x)
2.0
```

https://wiki.haskell.org/Converting_numbers

➢ Tipuri de baza

Boole Char String

Escape	Unicode	Character
\0	U+0000	null character
\a	U+0007	alert
\b	U+0008	backspace
\f	U+000C	form feed
\n	U+000A	newline (line feed)
\r	U+000D	carriage return
\t	U+0009	horizontal tab
\v	U+000B	vertical tab
\"	U+0022	double quote
\&	*n/a*	empty string
\'	U+0027	single quote
\\	U+005C	backslash

Bool
Valori: True, False
Operatori: == , /=
Operatii: &&, ||, not

Char : 'a', 'A' , '\n'

Modulul Data.Char :
chr, ord, toUpper, toLower

String = [Char]
"Sunt sir"
Operatii: ++, reverse, words, lines

Elementul k+1: str !! k

```
Prelude> let x= "a"++"\n"++"b"
Prelude> let y='a':'\n':'b':[]
Prelude> x == y
True
Prelude> lines x
["a","b"]
```

```
sirlung = "linia1\
\linia2\
\linia3\
\linia 4"
```

```
Prelude> "Sunt" == ['S','u','n','t']
True
Prelude> "Sunt"!!2
'n'
```

```
Prelude> :reload
Ok, modules loaded: Main.
*Main> sirlung
"linia1linia2linia3linia 4"
```

➢ Sistemul tipurilor

- tipuri compuse
- tipuri parametrizate
- clase de tipuri
- variabile tip
- constrangeri de tip
-
- signaturi de tipuri

➢ Tipuri compuse: tupluri (T1,…, Tn) n>1

```
Prelude> let x =(2::Int,'a',"a")
Prelude> :t x
x :: (Int, Char, [Char])
Prelude> type Mytuple = (Int, Char, [Char])
Prelude> let doi :: Mytuple -> Char; doi (x,y,z) =y
Prelude> doi x
'a'
```

Cu type definim sinonime pentru tipuri

Identificatorii de tipuri incep cu litera mare

fst si snd sunt definite numai pentru perechi

```
Prelude> fst (2,"s")
2
Prelude> snd (2,"s")
"s"
```

```
Prelude> :t ()
() :: ()
Prelude> :t (3)
(3) :: Num a => a
```

() se numeste **unit**

- **Liste** – secventa ordonata de elemente de acelasi tip
  constructorul pentru liste este   :

[] este lista vida
[1,2,3] si 1:2:3:[] reprezinta aceeasi lista

xs !! n este elementul din pozitia n a listei xs (pozitiile sunt numerotate incepand cu 0)

elem x xs este True daca x este element al listei xs(x \`elem\` xs)

Operatii cu liste:
length, ++, reverse, head, tail, take n, drop n , zip

<, >, <=, >=   le compara in ordine lexicografica
sum, product calculeaza suma, respective produsul elementelor
maximum, minimum

```
Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
Prelude> zip [1,2,3,4] ['a','b']
[(1,'a'),(2,'b')]
```

```
Prelude> :t reverse
reverse :: [a] -> [a]
Prelude> :t product
product :: Num a => [a] -> a
Prelude> :t minimum
minimum :: Ord a => [a] -> a
```

➢ Liste

```
Prelude> let suma x = sum [1 .. x]
Prelude> suma 4
10
Prelude> let inflistsuma' x = suma x : inflistsuma' (x+1)
Prelude> let inflistsuma = inflistsuma' 1
Prelude> let primelesume n = take n inflistsuma
Prelude> primelesume 10
[1,3,6,10,15,21,28,36,45,55]
```

sum este o functie definita in Prelude

```
Prelude> :t sum
sum :: Num a => [a] -> a
```

```
Prelude> :t permutations

<interactive>:1:1: Not in scope: ‘permutations’
Prelude> :m + Data.List
Prelude Data.List> :t permutations
permutations :: [a] -> [[a]]
Prelude Data.List> permutations "one"
["one","noe","eno","neo","eon","oen"]
```

- **Liste** – mai multe operatii in modulul Data.List

Exemplu: findIndex si lookup

```
Prelude> :m + Data.List
Prelude Data.List> let p :: String -> Bool; p "" = False; p (x:xs) = (x == 'A')
Prelude Data.List> p "Ion"
False
Prelude Data.List> p "Ana"
True
Prelude Data.List> findIndex p ["Radu", "Ion", "Ana", "Petre", "Alin"]
Just 2
Prelude Data.List> lookup 'a' [('a', 1), ('b',2), ('a',3),('c',6)]
Just 1
Prelude Data.List> lookup 'a' [('c', 1), ('b',2), ('b',3),('c',6)]
Nothing
```

- **List comprehension**: definirea listelor prin proprietati
  set comprehension {x| P(x)}
  list comprehension [E(x)| x <- [x1,…,xn], P(x)]

x <- [x1,…, xn], P(x)
x ia, pe rand, valorile x1,…,xn care verifica P(x)

```
Prelude> [x | x <- [1..20], x `mod` 3 ==2]
[2,5,8,11,14,17,20]
Prelude> [x*y| x <- [1,2], y<-[4,5] ]
[4,5,8,10]
Prelude> [(x,y)|x<-[1,2], y<-['a','b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
Prelude> [(x,y)|x<-[1,2], y<-"abc"]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]
```

```
Prelude> let fib ::[Integer]; fib = 0:1:[x+y | (x,y)<- zip fib (tail fib)]
Prelude> take 7 fib
[0,1,1,2,3,5,8]
```

Exemplu:

```
quick :: [Int]->[Int]
quick [] = []
quick (x:xs) = (quick ls) ++ [x] ++ (quick gt)
          where ls = [y | y<-xs, y <= x]
                   gt = [y | y<- xs, y > x]
```

```
Prelude> :t concat
concat :: [[a]] -> [a]
Prelude> concat [[1,2,3], [2,3],[1,4]]
[1,2,3,2,3,1,4]
Prelude> :m + Data.List
Prelude Data.List> :t nub        -- elimina duplicatele
nub :: Eq a => [a] -> [a]
Prelude Data.List> nub [1,2,3,1,2]
[1,2,3]
Prelude Data.List> let set = nub . concat
Prelude Data.List> set [[1,2,3], [2,3],[1,4]]
[1,2,3,4]
```

➢ **Functii**

O functie f care are ca argument o data de tip a si intoarce o data de tip b are tipul a -> b
pe care il declaram prin f :: a -> b

Tipul unei functii poate fi declarat explicit sau este dedus automat

Functiile pot fi definite

- ca functii anonime, folosind lambda-expresii \x1 … xn -> f(x1,… ,xn)
  ad = \ x y -> x+y

- folosind ecuatii <nume functie> <pattern> = <expresie>
  ad x y = x + y

  se apeleaza cu ad 2 3 sau 2 `ad` 3

operatorii . si $

f = g.h inseamna ca f(x)=g(h(x))
f $ x este egal cu f(x)

Functii

currying

```
Prelude> let ad :: (Int, Int) -> Int; ad (x,y) = x+y
Prelude> let adc = curry ad
Prelude> :t ad
ad :: (Int, Int) -> Int
Prelude> :t adc
adc :: Int -> Int -> Int
```

g : X x Y -> Z

f: X -> (Y -> Z)

f(x) :Y -> Z   pt. orice x din X

f(x)(y)=g(x,y)

partial application

```
Prelude> let power x y = x^y
Prelude> power 2 3
8
Prelude> let topower3 = (`power` 3)
Prelude> topower3 2
8
Prelude> let powerof3 = (3 `power`)
Prelude> powerof3 2
9
```

g: X x Y -> Z

x0 din X, fixat

f: Y->Z

f(y)=g(x0,y)

"Functions are first-class citizens“

Functiile pot fi atribuite unei variabile, transmise ca parametrii, returnate de o functie.

- functii anonime definite prin lambda expresii

```
Prelude> let myf  = \x y -> sqrt(x^2 + y^2)
Prelude> myf 1 2
2.23606797749979
Prelude> 1 `myf` 2
2.23606797749979
```

- functii atribuite unei variabile

```
Prelude> let myf  = \x y -> sqrt(x^2 + y^2)
Prelude> let nou = myf
Prelude> nou 1 2
2.23606797749979
```

➢ "Functions are first-class citizens"

Functiile pot fi atribuite unei variabile, transmise ca parametrii, returnate de o functie

- functii returnate de o functie

```
Prelude> let flip :: (a -> b -> c) -> (b -> a -> c); flip f x y = f y x
Prelude> let conc x y = x ++ y
Prelude> conc "111" "222"
"111222"
Prelude> flip conc "111" "222"
"222111"
```

notatie infix folosind `

```
-- conc "111" "222"
-- "111" `conc` "222"
```

- functii transmise ca parametru (unor functii de ordin superior – "higher-order functions")

```
Prelude> let myht :: (Floating a) => (a -> a) -> a -> a -> a; myht f x y = sqrt((f x) + (f y))
Prelude> let dbl x = x * x
Prelude> let myf x y = myht dbl x y
Prelude> myf 1 2
2.23606797749979
```

```
Prelude> :t dbl
dbl :: Num a => a -> a
Prelude> :t myht
myht :: Floating a => (a -> a) -> a -> a -> a
Prelude> :t myf
myf :: Floating a => a -> a -> a
```

➢ **Functii de ordin inalt predefinite:** **map, filter, foldl, foldr**

```
Prelude> let  triple x = x * x * x
Prelude> map triple [1,2,3]
[1,8,27]
Prelude> map triple (filter (>=2) [1,2,3])
[8,27]
Prelude> filter (>=2) [1,2,3]
[2,3]
Prelude> let sum' x y = (triple x) + (triple y)
Prelude> sum' 2 3
35
Prelude> foldl sum' 0 [1,2,3]
756
Prelude> foldl (+) 0 (map triple [1,2,3])
36
```

- Ranges

```
Prelude> [4..20]
[4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [3,4..20]
[3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [3,6..20]
[3,6,9,12,15,18]
Prelude> let x = [3..]
Prelude> take 20 x
[3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22]
```

```
Prelude> map (/2) [2,4..20]
[1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0]
```

```
Prelude> filter (< 3) [20, 4, 7, 34, 2, 0 ]
[2,0]
```

- Higher order function

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
Prelude> :t filter
filter :: (a -> Bool) -> [a] -> [a]
Prelude> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
Prelude> :t foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
```

foldl op z [x1,..., xn] calculeaza
((...(((z `op` x1)`op` x2) `op` x3)...)`op` xn)

➢ **Patterns**

```
Prelude> let x : y = [1,2,3]
Prelude> x
1
Prelude> y
[2,3]
Prelude> let (u,v,w)=('a', 3, [(1,'a'), (2, 'b')])
Prelude> w
[(1,'a'),(2,'b')]
Prelude> :t w
w :: Num t => [(t, Char)]
```

```
Prelude> let myhead [] = error "lista vida"; myhead (x:xs)=x
Prelude> myhead [1,2]
1
Prelude> myhead []
*** Exception: lista vida
Prelude> :t myhead
myhead :: [t] -> t
```

list pattern
tuple type
error function

```
Prelude> [1,2,3] == 1: 2: 3: []
True
Prelude> :t ('a', 1, [(1,'a'), (2, 'b')])
('a', 1, [(1,'a'), (2, 'b')])
  :: (Num t1, Num t) => (Char, t, [(t1, Char)])
Prelude> :t error
error :: [Char] -> a
```

- **Expresia** if … then … else
- **Garzi**
- **Expresia** case … of

mycontrol.hs

```
abs_if :: Integer -> Integer

abs_if x = if (x < 0) then (-x) else x


abs_guard :: Integer -> Integer

abs_guard x | x < 0 = -x
            |otherwise = x

ex_case :: (Integer, String) -> String

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

https://www.haskell.org/onlinereport/exps.html

➢ **Patterns & Guards**

```
schimb :: [Int] -> [Int]
schimb [] = []
schimb  (x:xs) = if (x == 1)
            then (0: schimb xs)
            else (x: schimb xs)


schimb1 :: [Int] -> [Int]
schimb1 [] = []
schimb1 (x:xs) | (x ==1) = (0: schimb1 xs)
               | otherwise = (x : schimb1 xs)
```

```
mesaj :: Int -> String
mesaj 0 = "liber"
mesaj 1 = "ocupat"
mesaj x | x >= 2 ="nedefinit"
        | otherwise = "imposibil"


mesaj1 :: Int -> String
mesaj1 x = case x of
        0 -> "liber"
        1 -> "ocupat"
        _
         | x >= 2 -> "nedefinit"
         |otherwise -> "imposibil"
```

Polymorphic types

```
Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Polimorfism parametric

"does the same thing to different types"

```
Prelude> let  e1 :: (a,b,c) -> a; e1 (x,y,z) = x
Prelude> let  e2 :: (a,b,c) -> b; e2 (x,y,z) = y
Prelude> let  e3 :: (a,b,c) -> c; e3 (x,y,z) = z
Prelude> e1 (2,"a",[3])
2
Prelude> e2 (2,"a",[3])
"a"
Prelude> e3 (2,"a",[3])
[3]
```

operatorul == este tot polimorfic,
dar in acest caz vorbim de
polimorfism ad-hoc

"does different things to different types"

➢ Tipuri noi definite cu data, type si newtype

data este folosit pentru a definit tipuri noi
type este folosit pentru a denumi tipuri deja existent
newtype este folosit pentru a defini tipuri echivalente cu tipuri deja existente

```
Prelude> data MyD = I Integer | C Char
Prelude> type MyDList = [MyD]
Prelude> newtype NeMyDList = Ne MyDList
```

constructor de tip   constructor de date

data Bool = True | False

➢ Tipuri noi definite cu data

```
data RGB = Rosu|Verde|Albastru
data Culoare = RGB | Rgb (Int, Int, Int)
data Modulo4 = V0|V1|V2|V3
```

➢ Sinonime pentru tipuri definite cu type

```
data Student = St String String
type CNP = Int
type Record = (Student,  CNP)
```

➢ **Tipuri parametrizate**

```
data Point a = Pt a a
```

```
*Main> let x = Pt (1:: Int) (2::Int)
*Main> :t x
x :: Point Int
```

➢ **Tipuri record (field labels)**

```
data Persoana = Pers  {nume::String, prenume::String}
```

```
*Main> let p = Pers {nume ="Ion", prenume="Popescu"}
*Main> nume p
"Ion"
*Main> let q = Pers "Ion" "Popescu"
*Main> :t q
q :: Persoana
*Main> prenume q
"Popescu"
```

Numele campurilor pot fi folosite ca functii selector pt. extragerea componentelor.

- **Tipuri definite recursiv**

```
data MyList a = Nil|
                Con a (MyList a)
```

```
*Main> let x = (Con 2 (Con 1 Nil))
*Main> :t x
x :: Num a => MyList a
```

```
data MyTree a =  NilTree |
                 Node a  (MyTree a) (MyTree a)
```

```
*Main> let x = Node "a" (Node "b" NilTree NilTree) NilTree
*Main> :t x
x :: MyTree [Char]
```

Sistemul tipurilor

clase de tipuri, variabile tip, constrangeri de tip, signaturi de tipuri

Mymod.hs

```haskell
module Mymod where

double :: (Num a) => a -> a
double x = x + x
```

```
*Mymod> :reload
[1 of 1] Compiling Mymod
Ok, modules loaded: Mymod.
*Mymod> double 5
10
*Mymod> double 4.5
9.0
```

Num
clasa care contine tipurile numerice:
Integer, Float, Double, …

(Num a) =>
constrangere de tip:
variabila tip a trebuie sa apartina clasei Num

double :: (Num a) => a -> a
signatura de tip

➢ **Tipuri parametrizate: Maybe a** (permite tratarea cazurilor de eroare)

```
Prelude> :m + Data.List
Prelude Data.List> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

Functia find intoarce primul element al listei care satisface proprietatea transmisa ca argument; Maybe a poate fi folosit pentru a semnala esecul

```
Data Maybe a = Just a | Nothing
```

```
Prelude Data.List> find (>3) [1,2]
Nothing
Prelude Data.List> find (>3) [1,3,4,5]
Just 4
```

Data.Maybe: fromJust

```
Prelude Data.List Data.Maybe> let x = find (>3) [3,4,5]
Prelude Data.List Data.Maybe> x
Just 4
Prelude Data.List Data.Maybe> :t x
x :: (Ord a, Num a) => Maybe a
Prelude Data.List Data.Maybe> let z = fromJust(x)+1
Prelude Data.List Data.Maybe> z
5
Prelude Data.List Data.Maybe> :t fromJust
fromJust :: Maybe a -> a
```

➢ Clasa de tipuri Show

```
*Main> let x = Node  1 (Node 2 NilTree NilTree) NilTree
*Main> :t x
x :: Num a => MyTree a
*Main> x

<interactive>:96:1:
    No instance for (Show (MyTree a0)) arising from a use of ‘print’
```

```
*Main> let x = Rosu
*Main> :t x
x :: RGB
*Main> x

<interactive>:178:1:
    No instance for (Show RGB)
```

```
data MyTree a =  NilTree | Node a  (MyTree a) (MyTree a)
                 deriving (Show)
```

```
data RGB = Rosu|Verde|Albastru
           deriving (Show)
```

```
*Main> let x = Node  1 (Node 2 NilTree NilTree) NilTree
*Main> x
Node 1 (Node 2 NilTree NilTree) NilTree
```

```
*Main> let x = Rosu
*Main> x
Rosu
```

➢ Clase de tipuri

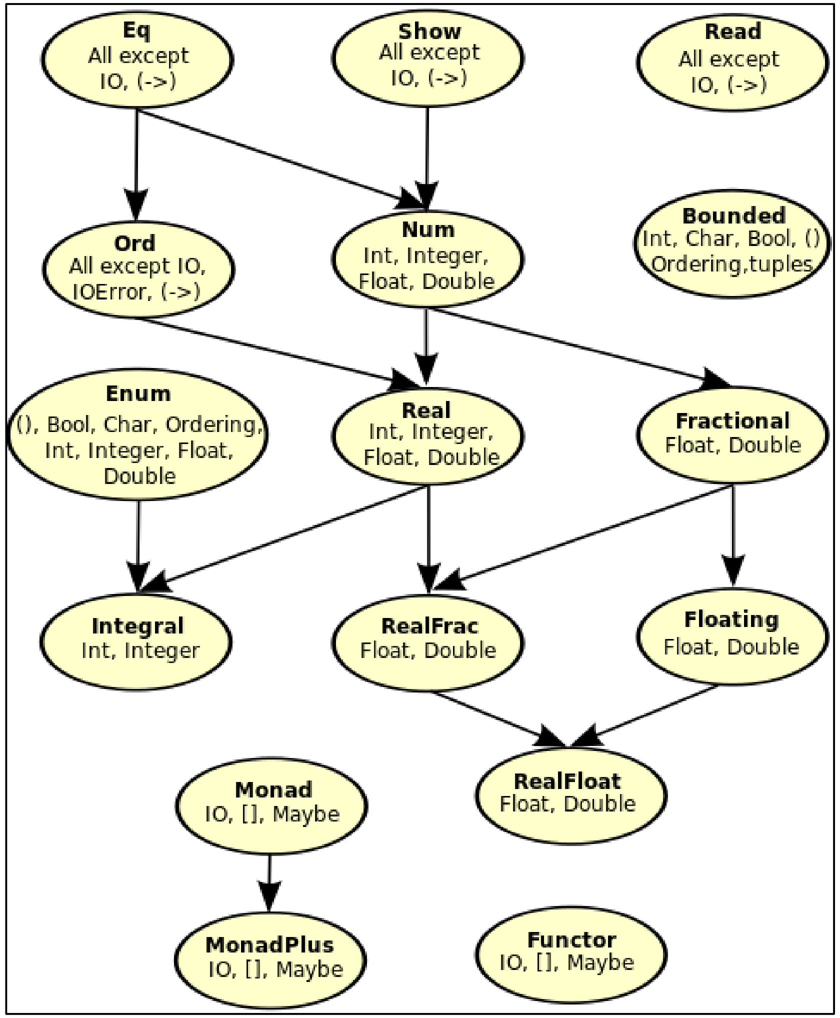
Diagrama claselor de tipuri predifinite in Haskell

http://en.wikibooks.org/wiki/Haskell/Classes_and_types

Instante ale clasei Show a sunt acele tipuri care pot fi convertite in String.
Clasa Show contine functia

```
show :: Show a => a -> String
```

```
instance Show RGB where
  show Rosu ="Red"
  show Verde = "Green"
  show Albastru = "Blue"
```

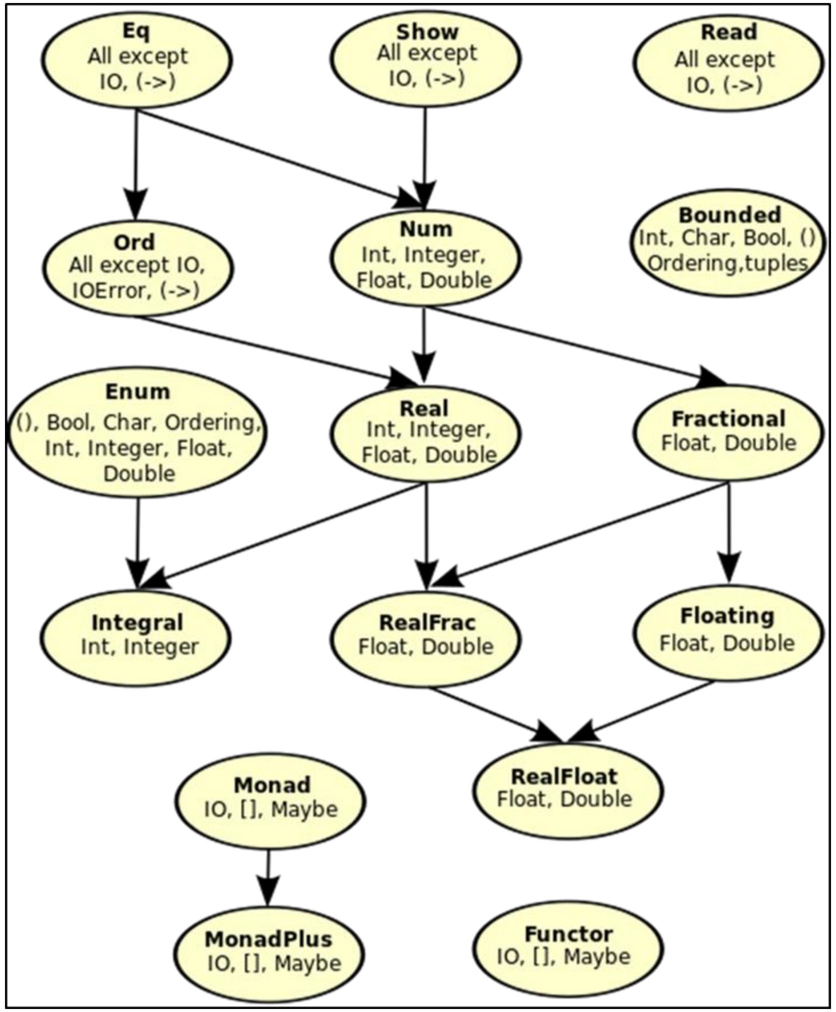



```
instance (Show a) => Show (MyTree a) where
    show NilTree = ""
    show (Node x  l r) = (show x)++"("++ (show l)++","++ (show r)++")"
```

- **Clasa de tipuri Read**

(este inversa lui Show)

```
Prelude> :t read
read :: Read a => String -> a
Prelude> read "3"
*** Exception: Prelude.read: no parse
Prelude> (read "34")::Int
34
Prelude> (read "3.4")::Float
3.4
Prelude> (read "3.478899988")::Double
3.478899988
Prelude> (read "3.478899988"):: Integer
*** Exception: Prelude.read: no parse
```



http://en.wikibooks.org/wiki/Haskell/Classes_and_types

```
Prelude> data Pers = Pers {nume :: String, prenume :: String}
Prelude> data Pers = Pers {nume :: String, prenume :: String} deriving (Eq, Read, Show)
Prelude> let a = Pers "Ana" "Ionescu"
Prelude> let b = Pers "Ana" "Popescu"
Prelude>  a == b
False
Prelude> show a
"Pers {nume = \"Ana\", prenume = \"Ionescu\"}"
Prelude> let str = "Pers \"Ana\" \"Popescu\""
Prelude> str
"Pers \"Ana\" \"Popescu\""
Prelude> read str::Pers
*** Exception: Prelude.read: no parse
Prelude> let str' = "Pers {nume=\"Ana\", prenume=\"Popescu\"}"
Prelude> read str' :: Pers
Pers {nume = "Ana", prenume = "Popescu"}
```

Exemplu: ierarhia claselor de tipuri numerice

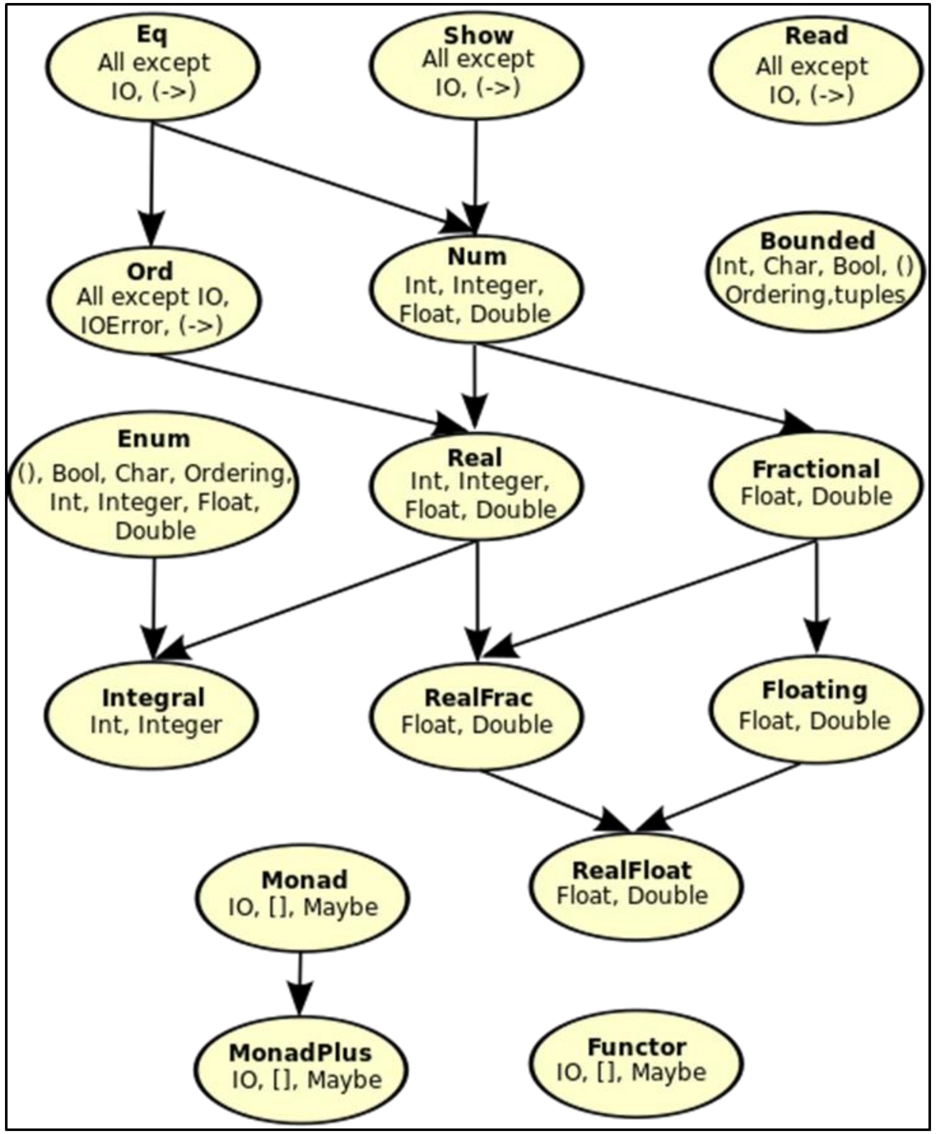
```
class Eq a where
   (==)  ::  a -> a -> Bool
   (/=)  ::  a -> a -> Bool     -- default method
   x /= y = not (x == y)


class (Eq a) => Ord a where
     (<),  (>),  (<=),  (>=) :: a -> a -> Bool
    …

class (Eq a, Show a) => Num a where
    (+),  (*) , (-) :: a -> a -> a
    abs :: a -> a
     ….

class (Eq a, Num a) => Real a where
   toRational :: a -> Rational
```

http://hackage.haskell.org/package/base-4.8.0.0/docs/Prelude.html



http://en.wikibooks.org/wiki/Haskell/Classes_and_types

➢ Clasa **Functor**

```
class Functor f  where
        fmap :: (a -> b) -> f a -> f b
```

poate fi gandita ca o generalizare a lui map

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

instance Functor [] where fmap = map
```

Proprietatile functorilor

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

```
instance Functor Maybe where
        fmap g Nothing = Nothing
        fmap g (Just x) = Just (g x)
```

```
Prelude Data.Maybe> fmap (*2) (Just 3)
Just 6
```

```
Prelude> data Point a = P a a deriving Show
Prelude> instance Functor Point where fmap g (P x y) = P (g x) (g y)
Prelude> fmap (*2) (P 2 3)
P 4 6
```

➢ Clasa Monad

O monada este o clasa m care are urmatoarele operatii:

fmap: (a-> b) -> (m a -> m b) -- Functor
return : a -> m a -- unit
join : m m a -> m a -- flattening

```
Prelude Data.Maybe Control.Monad> join (Just(Just 3))
Just 3
Prelude Data.Maybe Control.Monad> join [[1,2]]
[1,2]
Prelude Data.Maybe Control.Monad>
```

➢ Monade

O monada este o clasa m care are urmatoarele operatii:

fmap: (a-> b) -> (m a -> m b) -- Functor
return : a -> m a -- unit
join : m m a -> m a -- flattening

Categorie **C**
O *monada* este un functor M : **C** -> **C** astfel incat pentru fiecare obiect X din **C** exista doua morfisme:
unit_X : X -> M X
join_X : M M X -> X
care satisfac anumite proprietati.

Categoria **Hask** :
obiectele sunt tipuri,
morfismele sunt functii.

O monada in Haskell corespunde unei monade in categoria **Hask**

Monade

Monada m

```
fmap :: (a -> b) -> m a -> m b
return :: a -> m a
join :: m m a -> m a
```

Operatia >>= (bind)

```
>>= :: m a -> (a -> m b) -> m b
x  >>= g = join (fmap g  x)
                -- x  are tipul   m a
                -- g e are tipul a -> m b
```

https://www.haskell.org/tutorial/monads.html

Monada m

```
fmap :: (a -> b) -> m a -> m b
return :: a -> m a
join :: m m a -> m a
```

Operatia >>= (bind)

```
(>>=) :: m a -> (a -> m b) -> m b
x  >>= g = join (fmap g  x)
                 -- x  are tipul   m a
                 -- g e are tipul a -> m b
x >> y = x >>= \_ -> y
```

Operatia >=> (compunerea)

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)

f >=> g = \x ->  join ( fmap g (f x))
```

Operatia monadica >>=

m >>= g = join (fmap g m)

```
Prelude Control.Monad> data Point a = P a a deriving Show
Prelude Control.Monad> let pt x = P x x
Prelude Control.Monad> :t pt
pt :: a -> Point a
Prelude Control.Monad> let ptlist x =[P x x]
Prelude Control.Monad> :t ptlist
ptlist :: a -> [Point a]
Prelude Control.Monad> fmap ptlist [1]
[[P 1 1]]
Prelude Control.Monad> join (fmap ptlist [1])
[P 1 1]
Prelude Control.Monad> [1] >>= ptlist
[P 1 1]
```

Monada listelor

Operatia monadica >>

m >> k = m >>= _ -> k

```
Prelude Control.Monad> [1]>> [2]
[2]
Prelude Control.Monad> [1]>> [2]>> [3]
[3]
Prelude Control.Monad> let afiseaza = [P "o" "o"]
Prelude Control.Monad> [1]>> [2]>> [3] >> afiseaza
[P "o" "o"]
```

do notation

do e1; e2 = e1 >> e2

do x <- e1 ; e2 = e1 >>= \x -> e2

Monadele pot fi folosite pentru a programa in stil imperativ; codul "imperativ" ramane izolat de restul programului.

```
Prelude Control.Monad> let afiseaza1 = [P "o" "o"]
Prelude Control.Monad> let afiseaza2 = [P 1 1]
Prelude Control.Monad> afiseaza1 >> afiseaza2
[P 1 1]
Prelude Control.Monad> do afiseaza1; afiseaza2
[P 1 1]
Prelude Control.Monad> let afiseaza x =[P x x]
Prelude Control.Monad> [1] >>= afiseaza
[P 1 1]
Prelude Control.Monad> do x <- [1]; [P x x]
[P 1 1]
Prelude Control.Monad> do x <- [1]; afiseaza x
[P 1 1]
```

```
Prelude Control.Monad> :t afiseaza
afiseaza :: a -> [Point a]
Prelude Control.Monad> :t print
print :: Show a => a -> IO ()
Prelude Control.Monad> print "hello"
"hello"
```

Monada IO

➢ Monada IO

Intrarile si iesirile sunt valori de tipul IO a

```
Prelude> :t getLine
getLine :: IO String
Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

```
Prelude> :t getChar
getChar :: IO Char
Prelude> :t putChar
putChar :: Char -> IO ()
```

```
Prelude> :t print
print :: Show a => a -> IO ()
```

() unit este singura valoare a tipului () (singleton)

IO () este folosit atunci cand actiunile nu intorc valori semnificative

Valoarea de tip a dintr-o valoare de tip IO a se obtine folosind <-

```
s <- getLine
c <- getChar
```

➢ Actiuni IO

O valoare de tip (IO a) este o actiune care, daca este executata, produce o data de tip a.

Actiunile se comporta asemenea instructiunilor.
Secventele de actiuni de obtin folosind notatia do.

```
pg = do  putStrLn "Numele"
         s <- getLine
         putStrLn ("Hello " ++ s)
```

```
*Main> pg
Numele
Ioana
Hello Ioana
*Main> :t pg
pg :: IO ()
```

➢ Blocul do

In general un bloc do poate fi descris ca o secventa de Instructiuni, iar o instructiune poate fi:

- actiune, adica o expresie de tipul IO (de ex: getLine)
- o legatura <- (de ex: s <- getLine)
- o declaratie let (fara in)
- un apel al functiei return
- ultima linie trebuie sa fie o actiune sau return

```
pg = do
        putStrLn "introdu sirul"
        s <- getLine
        let n = length s     -- n este din clasa Num a
            t=n*n
        putStrLn (s ++ " are " ++ (show n) ++ " litere")
```

```
*Main> pg
introdu sirul
abcd
abcd are 4 litere
```

```haskell
getLine       :: IO String
getLine       =  do c <- getChar
                    if c == '\n'
                       then return ""
                       else do l <- getLine
                               return (c:l)
```

https://www.haskell.org/tutorial/io.html

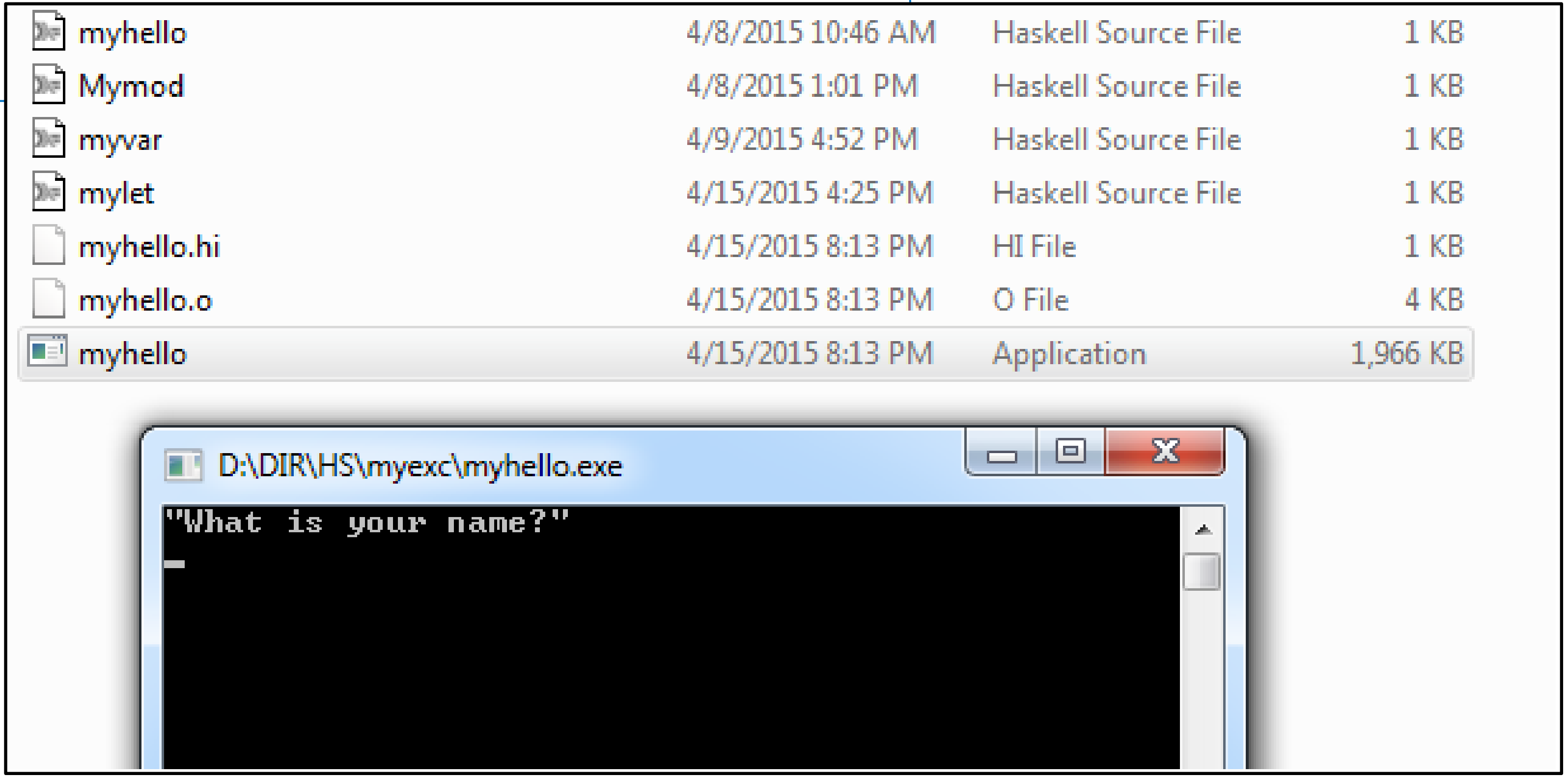
return :: String -> IO String

```
Prelude> :t return
return :: Monad m => a -> m a
```

- Programele pot fi compilate

A rula un program Haskell revine la a evalua functia main

```
Prelude>  :! ghc --make "myhello"
[1 of 1] Compiling Main             ( myhello.hs, myhello.o )
Linking myhello.exe ...
Prelude>
```




```
Prelude> :l myhello.hs
Ok, modules loaded: Main.
Prelude Main> :t main
main :: IO ()
Prelude Main>
```

Exemplu:

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

pg = do putStrLn "introdu  un numar"
        s <- getLine
        let n = (read s):: Int
            rez = take n primes
        print  rez

stop = do s<- getLine
          print s

main = do pg
          stop
```

- **Functii monadice:** **sequence, sequence_, mapM, mapM_, foldM, foldM_**

```
Prelude Control.Monad> [l1,l2,l3] <- sequence [getLine, getLine, getLine]
linia 1
linia 2
linia 3
Prelude Control.Monad> l2
"linia 2"
```

```
Prelude Control.Monad> mapM print [1,2,3]
1
2
3
[(),(),()]
Prelude Control.Monad> mapM_ print [1,2,3]
1
2
3
```

```
Prelude Control.Monad> let g = \x y -> (putStrLn  (x ++ show y)) >> (return $ (x ++ show y))
Prelude Control.Monad> let z = g "a" 4
Prelude Control.Monad> z
a4
"a4"
Prelude Control.Monad> foldM g "a" [1,2,3]
a1
a12
a123
"a123"
```