



Programare Funcțională

Prof. Gheorghe GRIGORAŞ

www.info.uaic.ro/~grigoras



Cursul 1 - Plan

- Informații generale despre curs
 - Cerințe, Logistica, Bibliografia
- Limbaje de programare
 - Functional vs. Imperativ
 - Pur vs. Impur
 - Lazy vs. Eager
 - Typed vs. Untyped
- Iстория на функционалните езици
- Introducere în Haskell
 - GHC, sesiuni, scripturi
- Tipurile de bază în Haskell
- Tipuri polimorfe, tipuri supraîncărcate



Cerințe

- Curs optional, anul II, sem II
 - Număr de credite: 5
 - Total ore: $5 \times 30 = 150$
 - Curs 28
 - Laborator 28
 - Activitate individuală:
 - Antrenament programare Haskell: 28
 - Pregătire teme laborator: 28
 - Parcursere bibliografie suplimentară 14
 - Pregătire examen scris: 24
- Prezența la toate laboratoarele
 - Evaluare săptămâna 8
 - 1 proiect(1 - 2 persoane) – săpt.9-15
 - 50% din nota finală
- Examen scris la sfârșit 50%



Bibliografie

- Graham Hutton, Programming in Haskell, Cambridge 2007, ([web](#))
- Richard Bird, Introduction to Functional Programming using Haskell, Prentice Hall Europe, 1998
- <https://wiki.haskell.org/Books>
- <https://wiki.haskell.org/Tutorials>
- <https://www.haskell.org/platform/>



Limbaje de programare

- Functional vs. Imperativ
- Pur vs. Impur
- Lazy vs. Eager
- Typed vs. Untyped



Functional vs. Imperativ

- Caracteristici ale stilului funcțional:
 - Structuri de date **persistente**: odată încărcate nu se mai schimbă
 - Metoda primară de calcul: **aplicarea funcțiilor**
 - Structura de control de bază: **recursia**
 - Utilizarea “din greu” a **funcțiilor de ordin înalt**: funcții ce au ca argument alte funcții și/sau ca rezultat funcții



Functional vs. Imperativ

- Caracteristici ale stilului imperativ:
 - Structuri de date **mutable**
 - Metoda primară de calcul: **atribuirea**
 - Structura de control de bază: **iterația**
 - Utilizarea **funcțiilor de ordinul întâi**
- Imperativ (C de exemplu):

```
suma = 0;  
for(i=1; i<=10; ++i)  
    suma = suma + i;
```

- Funcțional:

```
sum[1..10]
```



Quicksort in Haskell

quicksort :: Ord a => [a] -> [a]

quicksort [] = []

**quicksort (p:xs) = (quicksort lesser) ++ [p] ++
(quicksort greater)**

where

lesser = filter (< p) xs

greater = filter (>= p) xs



Quicksort in C

```
// To sort array a[] of size n: qsort(a,0,n-1)
void qsort(int a[], int lo, int hi){
    int h, l, p, t;
    if (lo < hi) {
        l = lo;    h = hi;    p = a[hi];
        do {
            while ((l < h) && (a[l] <= p))
                l = l+1;
            while ((h > l) && (a[h] >= p))
                h = h-1;
            if (l < h) {
                t = a[l]; a[l] = a[h]; a[h] = t;
            }
        } while (l < h);
        a[hi] = a[l]; a[l] = p;
        qsort( a, lo, l-1 );
        qsort( a, l+1, hi );
    }
}
```



Avantaje/Dezavantaje

- Functional programs are often easier to **understand**.
- Functional programs tend to be much more **concise, shorter by a factor of two to ten usually, than their imperative counterparts**.
- Polymorphism enhances re-usability: `qsort` program in Haskell will not only sort lists of integers (*C version*), but also lists of *anything for which it is meaningful to have "less-than" and "greater-than" operations*.
- Store is allocated and initialized implicitly, and recovered automatically by the garbage collector.

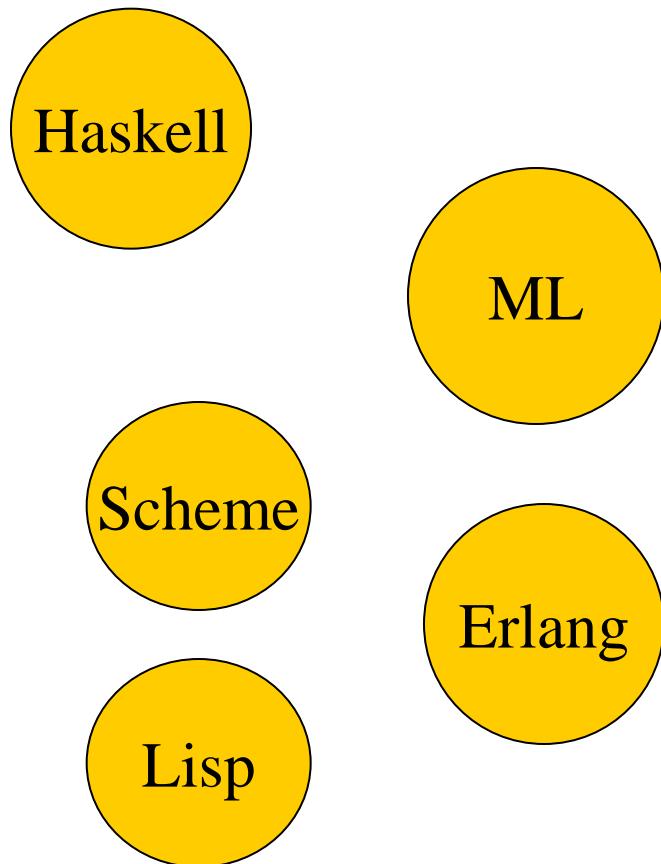


Avantaje/Dezavantaje

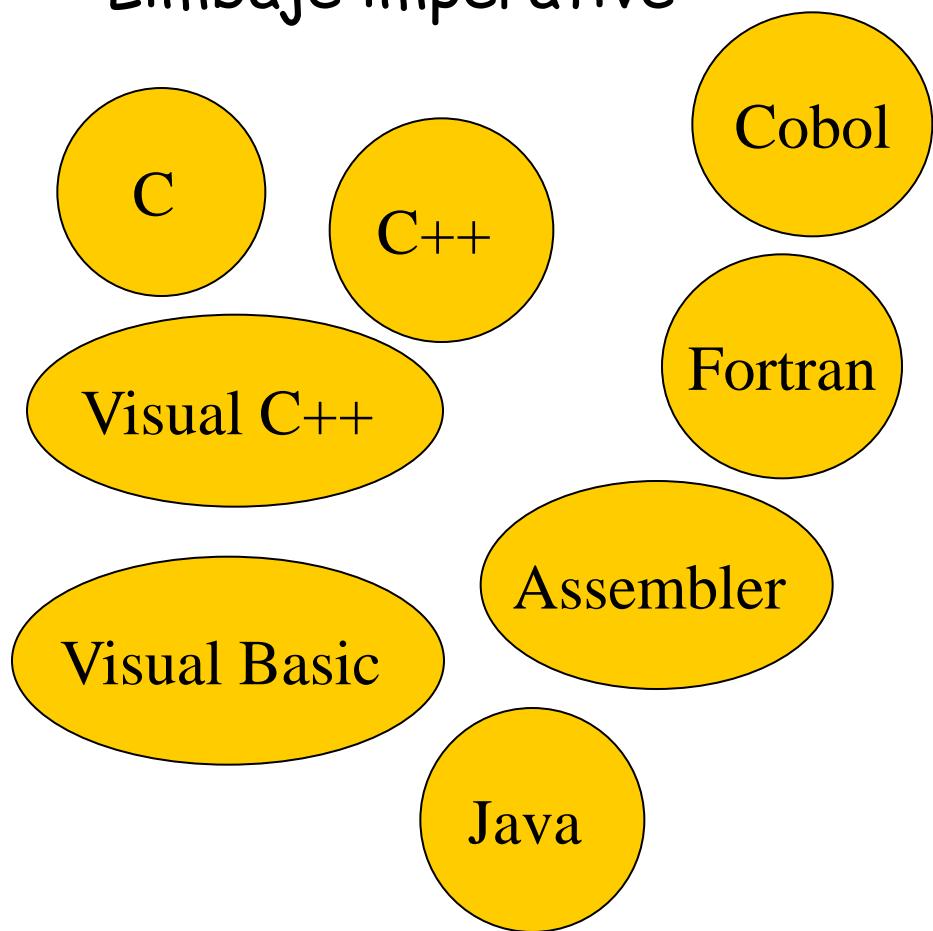
- The C quicksort uses an extremely ingenious technique, invented by Hoare, whereby it sorts the array *in place*; *that is, without using any extra storage.*
- As a result, it runs quickly, and in a small amount of memory.
- In contrast, the Haskell program allocates quite a lot of extra memory behind the scenes, and runs rather slower than the C program.
- In applications where performance is required at any cost, or when the goal is detailed tuning of a low-level algorithm, an imperative language like C would probably be a better choice than Haskell



Limbaje funcționale



Limbaje imperative





Aplicații industriale implementate funcțional

<http://homepages.inf.ed.ac.uk/wadler/realworld/index.html>

Intel (microprocessor verification)

Hewlett Packard (telecom event correlation)

Ericsson (telecommunications)

Carlstedt Research & Technology (air-crew scheduling)

Legasys (Y2K tool)

Hafnium (Y2K tool)

Shop.com (e-commerce)

Motorola (test generation)

Thompson (radar tracking)

https://wiki.haskell.org/Haskell_in_industry



Pur vs. Impur

- Un limbaj funcțional ce are doar caracteristicile stilului funcțional (și nimic din stilul imperativ) este limbaj funcțional pur. Haskell este limbaj funcțional pur.
- Limbaje ca: Standard ML, Scheme, Lisp combină stilul funcțional cu o serie de caracteristici imperative; acestea sunt limbaje impure.



Lazy vs. Eager

- Limbajele funcționale se împart în două categorii:
 - Cele care evaluatează funcțiile (expresiile) în mod **lazy**, adică argumentele unei funcții sunt evaluate numai dacă este necesar
 - Cele care evaluatează funcțiile(expresiile) în mod **eager**, adică argumentele unei funcții sunt tratate ca și calcule precedente funcției și sunt evaluate înainte ca funcția să poată fi evaluată



- Haskell este un limbaj lazy:

```
Prelude> let f x = 10
```

```
Prelude> f(1/0)
```

```
10
```

```
Prelude> 1/0
```

```
Infinity
```

```
Prelude> let poz_ints = [1..]
```

```
Prelude> let one_to_ten = take 10 poz_ints
```

```
Prelude> one_to_ten
```

```
[1,2,3,4,5,6,7,8,9,10]
```



Typed vs. Untyped

- Limbajele pot fi categorisite astfel:
 - Puternic tipizate (strongly typed): împiedică programele să acceseze date private, să corupă memoria, să blocheze sistemul de calcul etc.
 - Slab tipizate (weakly typed)
 - Static tipizate (statically typed): consistența tipurilor este verificată la compilare
 - Dinamic tipizate (dynamically typed): verificarea consistenței se face la execuție



Typed vs. Untyped

- Limbajul Haskell este puternic tipizat
- Orice expresie ce apare într-un program Haskell are un tip ce este cunoscut la compilare (statically typed).
- Nu există mecanisme care să “distrugă sistemul de tipuri”.
- Ori de câte ori se evaluatează o expresie, de tip întreg de exemplu, rezultatul va fi de același tip, întreg.



Haskell este liber de **side effects**!

- Nu are asignări
- Nu are variabile mutabile
- Nu are tablouri mutabile
- Nu are înregistrări mutabile
- Nu are stări ce pot fi actualizate



Avantajele alegerii limbajului Haskell

- Haskell este un limbaj de nivel foarte înalt (multe detalii sunt gestionate automat).
- Haskell este expresiv și concis (se pot obține o sumedenie de lucruri cu un efort mic).
- Haskell este adekvat în a manipula date complexe și a combina componente.
- Cu Haskell programatorul câștigă timp: “programmer-time” este prioritar lui “computer-time”



Istoric al limbajelor funcționale

- 1920 – 1940, Alonzo Church(1903 – 1995) și Haskell Curry(1900 – 1982) dezvoltă “Lambda Calculul”, o teorie matematică a funcțiilor

<http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Church.html>





Istoric al limbajelor funcționale

- 1970-80, David Turner
 - Limbaje funcționale lazy
 - Limbajul Miranda (“admirable”)
- 1987: începe dezvoltarea limbajului Haskell de către un comitet internațional (numele vine de la Haskell Curry) – limbaj funcțional lazy standard
- 2003: este publicat raportul cu definiția limbajului Haskell 98 (“the culmination of fifteen years of revision and extensions”)



Implementări Haskell

- <https://www.haskell.org/platform/>
- **Platforma Haskell:**
 - Current release: [2014.2.0.0](#)
 - The Haskell Platform is the easiest way to get started with programming Haskell.
 - The Haskell Platform contains only stable and widely-used tools and libraries, drawn from a pool of thousands of [Haskell packages, ensuring you get the best from what is on offer.](#)
 - The Haskell Platform ships with advanced features such as multicore parallelism, thread sparks and transactional memory



Introducere în Haskell; GHC

- GHC, <http://hackage.haskell.org/platform/>

```
GHCI, version 7.0.4: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```



Sesiuni

- La lansarea sistemului se încarcă o bibliotecă Prelude.hs în care sunt definite funcțiile uzuale:
- Sesiune: secvența de interacțiuni utilizator – sistem

```
Prelude> 56-3*(24+4/2) / (4-2)
17.0
Prelude> 3^21
10460353203
Prelude> 22^33
199502557355935975909450298726667414302359552
Prelude> let x = 24 in (x + 14)*(x - 14)
380
Prelude> :{
Prelude| let y = 33
Prelude| in 12 + y*3
Prelude| :}
111
```



Sesiuni

```
Prelude> [1,2,3] ++ [5,6]
[1,2,3,5,6]
Prelude> head [1,2,3,4]
1
Prelude> tail [1,2,3,4]
[2,3,4]
Prelude> take 2 [1,2,3,4]
[1,2]
Prelude> drop 2 [1,2,3,4]
[3,4]
Prelude> [1,2,3,4,5] !! 2
3
Prelude> [1,2,3,4,5] !! 9
*** Exception: Prelude.(!!): index too large

Prelude> sum [1..10]
55
```



Scripturi (fișiere sursă)

- O listă de definiții constituie un script:
 - Se editează un fișier cu definiții

```
main = print (fac 20)
```

```
fac 0 = 1
```

```
fac n = n * fac (n-1)
```

- Se salvează (de exemplu Main.hs)
- Interpretare:
 - Se încarcă acest fișier:
:load Main.hs sau :l Main.h
 - Se invocă oricare din numele definite în fișier



Scripturi

```
main = print (fac 20)
fac 0 = 1
fac n = n * fac (n-1)
```

```
Prelude> :l Main.hs
[1 of 1] Compiling Main           ( Main.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
2432902008176640000
*Main> fac(5)
120
*Main> fac(20)
2432902008176640000
```



Scripturi

```
main = print (fac 20)
fac 0 = 1
fac n = n * fac (n-1)
```

- Compilare:

```
Prelude> :!ghc -o main Main.hs
```

```
Prelude> :! main
```

```
2432902008176640000
```



Scripturi – modificare + reîncărcare

- Se adaugă la fișierul existent (Main.hs) alte definiții:

...

```
smaller x y = if x < y then x else y
biger x y = if x > y then x else y
```

- Se salvează
- Se reîncarcă acest fișier:

```
*Main> :r
Ok, modules loaded: Main.
*Main> bigger 55 66
66
*Main> smaller 5 6
5
```



Reguli sintactice

- În Haskell funcțiile au prioritatea cea mai mare:
 $f\ 2 + 3$ înseamnă $f(2) + 3$
- Apelul funcțiilor se poate face fără a pune argumentele în paranteză:

$f(a)$	$f\ a$
$f(x, y)$	$f\ x\ y$
$f(g(x))$	$f(gx)$
$f(x, g(y))$	$f\ x\ (g\ y)$
$f(x)g(y)$	$f\ x\ * g\ y$



Reguli sintactice

- Numele funcțiilor și a argumentelor trebuie să înceapă cu literă mică sau cu `_`: myF, gMare, x, uu, `_x`, `_y`
- Se recomandă ca numele argumentelor de tip listă să aibă sufixul `s`: xs, ns, us, ps, a13s
- Într-o secvență de definiții toate definițiile trebuie să înceapă pe aceeași coloană:

`a = 30`

`b = 21`

`c = 111`

~~`a = 30`~~

~~`b = 21`~~

~~`c = 111`~~

~~`a = 30`~~

~~`b = 21`~~

~~`c = 111`~~



Cuvinte cheie (21)

Erlang - 28, OCaml - 48, Java - 50,
C++ - 63, Miranda - 10

case	class	data
default	deriving	do
else	if	import
in	infix	infixl
infixr	instance	let
module	newtype	of
then	type	where



Expresii, Valori

- Expresiile denotă valori
- O expresie poate conține:
 - Numere
 - Valori de adevăr
 - Caractere
 - Tuple(n-uple)
 - Funcții
 - Liste
- O valoare are mai multe reprezentări
- reprezentări ale lui 12:

12

2+10

3*3+3



Evaluare (reducere, simplificare)

- O expresie este evaluată prin reducerea sa la forme mai simple echivalente
- Evaluatorul limbajului funcțional scrie forma canonică a expresiei care denotă o valoare
- Obținerea formei canonice se face printr-un proces de reducere a expresiilor
- Reducerea se poate face în mai multe moduri: rezultatul trebuie să fie același



Evaluare (reducere, simplificare)

square (2+7)

def +

square (9)

def square

9*9

def *

81

square (2+7)

def square

(2+7) * (2+7)

def +

9* (2+7)

def +

9*9

def *

81



Evaluare (reducere, simplificare)

- Considerăm scriptul:

```
trei x = 3  
infinit = infinit + 1
```

```
trei infinit  
def infinit  
trei(infinit + 1)  
def infinit  
trei((infinit + 1)+1)  
...
```

```
trei infinit  
def trei  
3
```

- Strategia lazy evaluation asigură terminarea procesului atunci când acest lucru este posibil



Tipuri

- Tip : colecție de valori împreună cu operațiile ce se pot aplica acestora
 - **Bool** conține **False** și **True**
 - **Bool** \rightarrow **Bool** conține toate funcțiile cu argumente din **Bool** și valori în **Bool**
- Se utilizează notația **v** $:::$ **T** pentru a exprima că v are tipul T
- Orice expresie are un tip ce se determină, după reguli precise, înainte de a evalua expresia
- Procesul de determinare a tipului este numit “type inference”



Tipuri

- Tipul unei expresii se poate afla în Haskell:

```
Prelude> :type 4
4 :: (Num t) => t
Prelude> :type 4.54
4.54 :: (Fractional t) => t
Prelude> :type False
False :: Bool
Prelude> :t "alpha" ++ "beta"
"alpha" ++ "beta" :: [Char]
Prelude> :t [1,2,3]
[1,2,3] :: (Num t) => [t]
Prelude> :t ['a', 'b', 'c']
['a', 'b', 'c'] :: [Char]
```



Tipurile de bază în Haskell

- Bool
 - Valori: **False True**
 - Operații: **&& || not**
- Char
 - Valori: caractere
 - Informații despre tipul char:
Prelude> :i Char



Tipurile de bază în Haskell

- Operații ale tipului Char

```
Prelude> 'a' == 'b'  
False  
Prelude> 'a' /= 'b'  
True  
Prelude> 'a' < 'b'  
True  
Prelude> 'a' >= 'b'  
False  
Prelude> succ 'a'  
'b'  
Prelude> pred 'a'  
'`'  
Prelude> pred 'b'  
'a'  
Prelude> toEnum 57 :: Char  
'9'  
Prelude> fromEnum 'b'  
98
```



Tipurile de bază în Haskell

- String
 - Valori: liste de caractere
 - Operații: (vezi liste)

```
Prelude> "99" ++ "a"  
"99a"  
  
Prelude> head "alpha"  
'a'  
  
Prelude> take 4 "abcdef"  
"abcd"  
  
Prelude> drop 4 "abcdef"  
"ef"
```



Tipurile de bază în Haskell

- Int: - $2^{31}..2^{31}-1$ (întregi precizie fixă)
- Integer (întregi precizie arbitrară)

```
Main> 2^31::Int
```

```
-2147483648
```

```
Main> 2^31
```

```
2147483648
```

```
Main> 2^31::Integer
```

```
2147483648
```

```
Main> 2^99
```

```
633825300114114700748351602688
```

– Operații: + - * ...



Tipurile de bază în Haskell

- Float, Double
 - Numere reale în simplă(dubla) precizie: 2.2331, +1.0, 3.141
 - Numărul de cifre semnificative:
 - Virgula fixă: 7 – Float, 16 – Double
 - Virgula mobilă: 1+7, 1+16

```
Prelude> sqrt 4565
67.56478372643548
Prelude> sqrt 4565::Float
67.56478
Prelude> sqrt 456554545454::Float
675688.2
Prelude> sqrt 456554545454
675688.2013576973
Prelude> sqrt 456554545454456667777
2.1367137043938683e10
Prelude> sqrt 456554545454456667777::Float
2.1367138e10
```



Tipul Listă în Haskell

- Lista: o secvență de elemente de același tip
- Sintaxa pentru acest tip: [T], unde T este tipul elementelor

```
Prelude> :t [1,2,3]
[1,2,3] :: (Num t) => [t]
Prelude> :t [True, True, False]
[True, True, False] :: [Bool]
Prelude> :t ["True","True","False"]
["True","True","False"] :: [[Char]]
Prelude> :t [[1], [1,2,3], []]
[[1], [1,2,3], []] :: (Num t) => [[t]]
Prelude> :t [[[[]]]]
[[[[]]]] :: [[[a]]]
```

- [] lista vidă, [1], ["unu"], [[]] liste singleton



Tipul n - uplă în Haskell

- O secvență finită de componente de diferite tipuri; componentele, despărțite prin virgulă, sunt incluse în paranteze rotunde
- Reprezentare: (T_1, T_2, \dots, T_n)
- Numărul componentelor = aritatea tuplei
 - Tupla () – tupla vidă, $(T) = T$
 - Perechi, triplete
 - Nu există restricții asupra tipului componentelor

```
Prelude> :t ()  
() :: ()  
  
Prelude> :t (1)  
(1) :: (Num t) => t  
  
(2) Prelude> :t ('a', "a", False, 3.4)  
('a', "a", False, 3.4) :: (Fractional t) => (Char, [Char],  
Bool, t)
```



Tipul funcție în Haskell

- Funcție în sens matematic, de la tipul T_1 la tipul T_2
- $T_1 \rightarrow T_2$ notează tipul funcțiilor de la T_1 la T_2
- Convenție Haskell: definiția unei funcții este precedată de tipul său:

```
pi      :: Float
pi = 3.14159
square :: Integer -> Integer
square x = x*x
plus   :: (Int, Int) -> Int
plus(x, y) = x + y
Suma   :: Int -> Int -> Int
Suma x y = x + y
```



Funcții curry

- O funcție cu $n > 1$ argumente este definită ca funcție cu un argument ce are ca valori o funcție cu $n-1$ argumente

```
plusc :: Int -> (Int -> Int)
```

```
plusc x y = x + y
```

```
mult :: Int -> (Int -> (Int -> Int))
```

```
mult x y z = x*y*z
```

```
twise :: (Int -> Int) -> (Int -> Int)
```

```
twise f x = f(fx)
```

```
twise f = f.f
```



Funcții curry

- Convenții pentru funcții curry
 - “Operatorul” \rightarrow în tipuri are asociativitatea dreapta
 $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
înseamnă
 $\text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))))$

`mult x y z` înseamnă `((mult x)y)z`



Tipuri polimorfe

- Funcții ce sunt definite pentru mai multe tipuri: lungimea unei liste se calculează indiferent de tipul elementelor listei
- Variabile tip: a, b, c, ...

```
Prelude> :t length
length :: [a] -> Int
Prelude> length "abcdef"
6
Prelude> length [1,2,3,4,5,6,7,8,9]
9
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t snd
snd :: (a, b) -> b
Prelude> :t head
head :: [a] -> a
```

- Un tip ce conține una sau mai multe **variabile tip** se numește **tip polimorf**



Tipuri supraîncărcate

- Operatori ce se aplică la mai multe tipuri (+ , * , / , < , > , ...)
- Tipul acestora conține **variabile tip** supuse unor **constrângeri**
- Astfel de tipuri se numesc **tipuri supraîncărcate**

```
Prelude> :t (+)
(+) :: (Num a) => a -> a -> a
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> :t (<)
(<) :: (Ord a) => a -> a -> Bool
Prelude> :t (/=)
(/=) :: (Eq a) => a -> a -> Bool
```



Cursul 2 - Plan

- Clase(de tipuri) în limbajul Haskell
- Definirea funcțiilor
 - Prin folosirea unor funcții existente
 - Expresii condiționale
 - Ecuații cu gardă
 - Potrivire şablonane
 - Şablonane tuple
 - Şablonane listă
 - Şablonane întregi

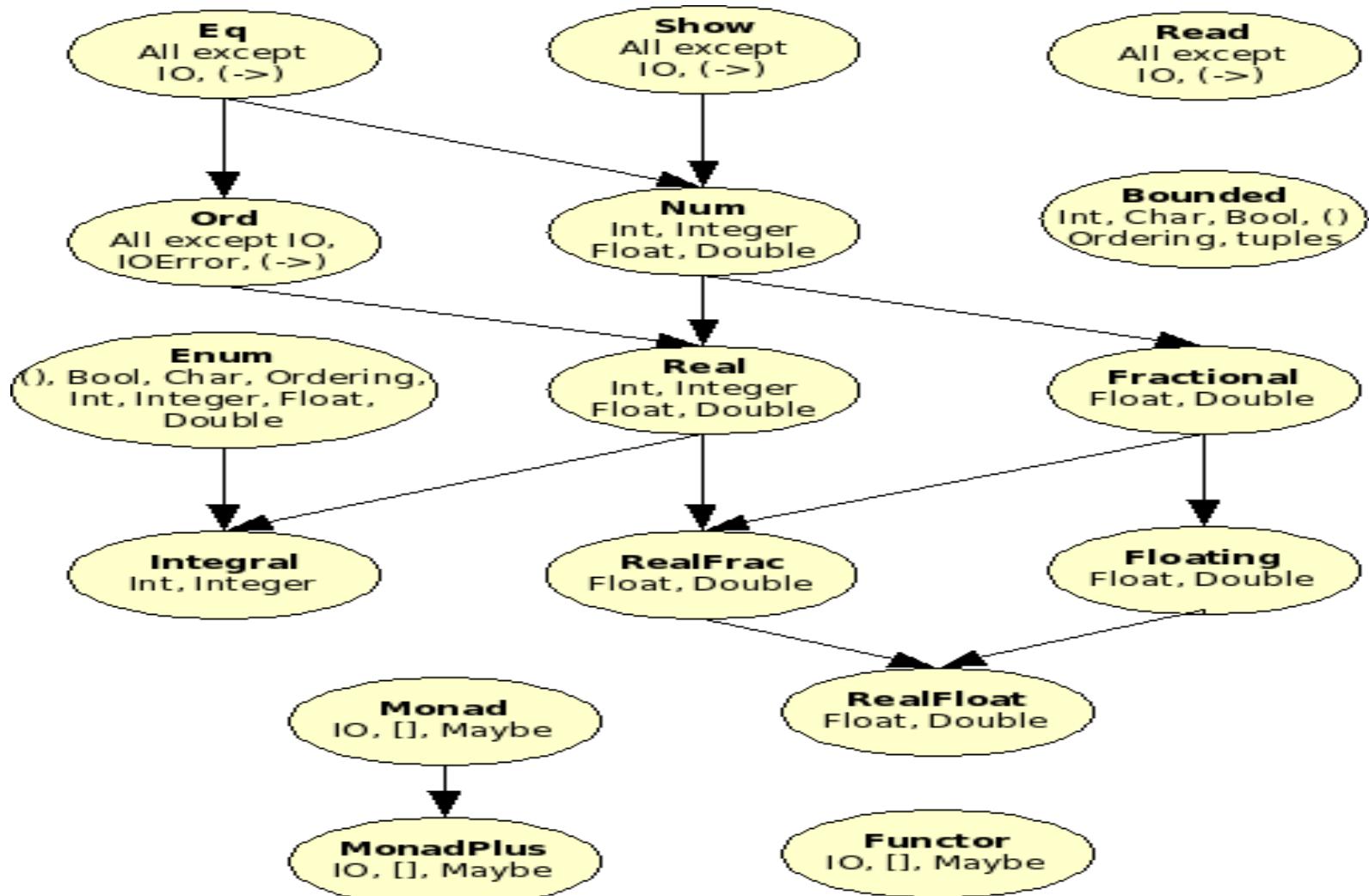


Clasele de bază

- Tip – colecție de date împreună cu operații
 - Bool, Char, String, Int, Integer, Float
 - Tipul listă
 - Tipul tuplă
 - Tipul funcție
- Clasă – colecție de tipuri care suportă operații supraîncărcate numite *metode*
 - Eq, Ord, Show, Read, Num, Integral, Fractional



Clase în Haskell





Eq – clasa tipurilor cu egalitate

- Tipuri ce au valori care pot fi comparate pentru egalitate și neegalitate
- Metodele:
 - `(==) :: Eq a => a -> a -> Bool`
 - `(/=) :: Eq a => a -> a -> Bool`
- Tipuri componente:
 - Tipuri de bază: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`
 - Tipuri listă și tuple cu elemente (componente) tipuri din clasa `Eq`



Ord – clasa tipurilor ordonate

- Tipuri ce sunt instanțe ale clasei Eq și care au valorile total ordonate
- Metodele:
 - `(<)` :: Ord a => a -> a -> Bool
 - `(<=)` :: Ord a => a -> a -> Bool
 - `(>)` :: Ord a => a -> a -> Bool
 - `(>=)` :: Ord a => a -> a -> Bool
 - `min` :: Ord a => a -> a -> a
 - `max` :: Ord a => a -> a -> a
- Tipuri componente:
 - Tipuri de bază: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`
 - Tipuri listă și tuple cu elemente (componente) tipuri din clasa `Ord`



Exemple

```
Prelude> True > False
```

```
True
```

```
Prelude> "babababab" < "ababababa"
```

```
False
```

```
Prelude> ['a' , 'b' , 'c'] >= ['x']
```

```
False
```

```
Prelude> (1,2,3) < (1,2,4)
```

```
True
```

```
Prelude> min [1,2,-4,5] [1,2,3]
```

```
[1,2,-4,5]
```

```
Prelude> max (True, False) (False, False)
```

```
(True, False)
```

```
Prelude> sqrt 777 >= 44
```

```
False
```



Show

- Tipuri ce conțin valori exprimabile prin siruri de caractere
- Metodele:
 - **show :: Show a => a -> String**
- Tipuri componente:
 - Tipuri de bază: **Bool, Char, String, Int, Integer, Float, Double**
 - Tipuri listă și tuple cu elemente (componente) tipuri din clasa **Show**



Exemple

```
Prelude> show 76890
```

```
"76890"
```

```
Prelude> show [[12,33],[1],[1,2,3]]
```

```
"[[12,33],[1],[1,2,3]]"
```

```
Prelude> show ("True", True)
```

```
"(\"True\",True)"
```

```
Prelude> show (True, True)
```

```
"(True,True)"
```

```
Prelude> show ('a',777)
```

```
"('a',777)"
```

```
Prelude> show (44+99)
```

```
"143"
```



Read – clasa tipurilor ce pot fi “citite”

- Tipuri ce conțin valori care pot fi convertite din siruri de caractere
- Metodele:
 - `read :: Read a => String -> a`
- Tipuri componente:
 - Tipuri de bază: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`
 - Tipuri listă și tuple cu elemente (componente) tipuri din clasa `Read`



Exemple

```
Prelude> read "123":Int  
123  
Prelude> read "123"+ read "123"  
246  
Prelude> read "123.65" ::Float  
123.65  
Prelude> read "123.65" ::Int  
*** Exception: Prelude.read: no parse  
Prelude> read "123.65" ::Double  
123.65  
Prelude> read "[1, 2]" ::[Int]  
[1,2]  
Prelude> read "[True, False, True, True]":Bool  
[True,False,True,True]  
  
Prelude> read "abcd":String  
*** Exception: Prelude.read: no parse  
Prelude> read "\"abcd\"":String  
"abcd"
```



Num – clasa tipurilor numerice

- Tipuri ce sunt instanțe ale claselor **Eq** și **Show** și au valori numerice (**Int**, **Integer**, **Float**, **Double**)
- Metodele:

(+) :: Num a => a -> a -> a

(-) :: Num a => a -> a -> a

(*) :: Num a => a -> a -> a

negate :: Num a => a -> a

abs :: Num a => a -> a

signum :: Num a => a -> a

fromInteger :: Integer -> a



Exemple

```
Prelude> fromInteger 33 :: Integer  
33  
Prelude> fromInteger 33 :: Double  
33.0  
Prelude> let x = fromInteger 22 :: Double  
Prelude> x  
22.0  
Prelude> :t x  
x :: Double  
Prelude> fromInteger 3333333333333333 :: Int  
553997653  
Prelude> fromInteger 3333333333333333 :: Double  
3.33333333333333e19
```



Real - clasa tipurilor numerice

- Tipurile `Int`, `Integer`, `Float`, `Double`
- Cum se definește:

```
Prelude> :i Real
```

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

```
    -- Defined in GHC.Real
```

```
instance Real Integer -- Defined in  
GHC.Real
```

```
instance Real Int -- Defined in GHC.Real
```

```
instance Real Double -- Defined in  
GHC.Float
```

```
instance Real Float -- Defined in  
GHC.Float
```



Exemple

```
Prelude> toRational 12
12 % 1
Prelude> toRational 12.4
6980579422424269 % 562949953421312
Prelude> toRational 1.5
3 % 2
Prelude> toRational 3
3 % 1
Prelude> toRational 3.5
7 % 2
Prelude> toRational (12.4::Float)
6501171 % 524288
Prelude> (12.4::Float)
12.4
Prelude> toRational (12.4::Double)
6980579422424269 % 562949953421312
```

Integral – clasa tipurilor întregi

- Tipuri `Int`, `Integer`

- Metodele:

`div :: Integral a => a -> a -> a`

`mod :: Integral a => a -> a -> a`

- Metodele pot fi utilizate și ca operatori infix dacă se scriu ``div``, ``mod`` (atenție: ``` și `nu '`)



Exemple

```
Hugs.Base> 22 `mod` 5
```

```
2
```

```
Hugs.Base> 22 `mod` 3
```

```
1
```

```
Hugs.Base> 22 `div` 3
```

```
7
```

```
Hugs.Base> div 22 3
```

```
7
```

```
Hugs.Base> mod 22 3
```

```
1
```



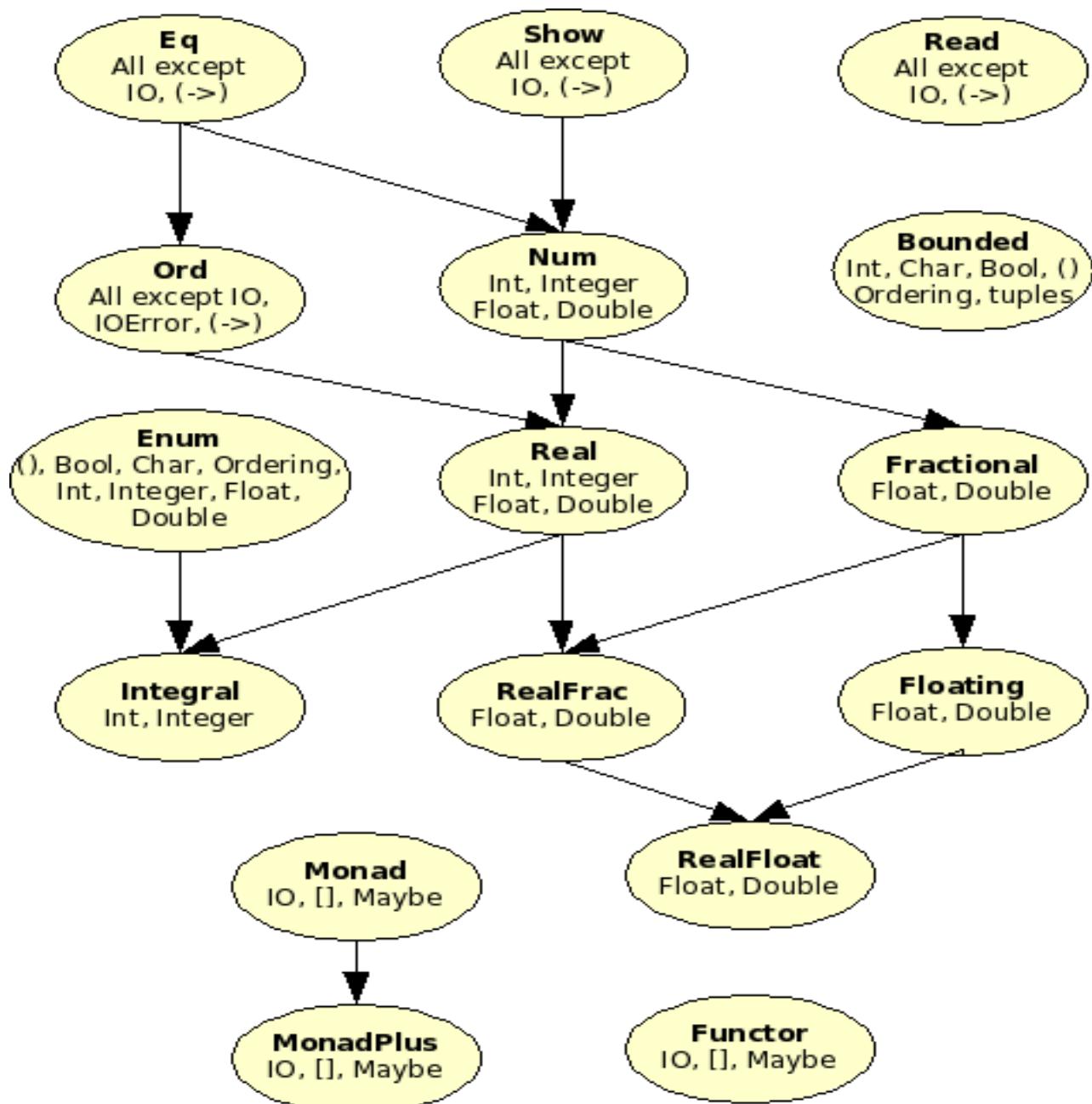
Fractional – clasa tipurilor reale

- Tipuri ce sunt instanțe ale clasei **Num** și au valori neîntregi (**Float**, **Double**)
- Metodele:
`(/) :: Fractional a => a -> a -> a`
`recip :: Fractional a => a -> a`



Exemple

```
Hugs.Base> 7/2
3.5
Hugs.Base> 7./2
ERROR - Undefined variable "./"
Hugs.Base> 7.0/2
3.5
Hugs.Base> 7.3/2
3.65
Hugs.Base> recip 2
0.5
Hugs.Base> recip 1
1.0
Hugs.Base> recip 1.5
0.66666666666667
```





Definirea funcțiilor

- Prin folosirea unor funcții existente:

```
isDigit :: Char -> Bool
```

```
isDigit c = c >= '0' && c <= '9'
```

```
estePar :: Integral a => a -> Bool
```

```
estePar n = n `mod` 2 == 0
```

```
rupela :: Int -> [a] -> ([a], [a])
```

```
rupela n xs = (take n xs, drop n xs)
```



Exemple

```
Prelude> :l c2test
[1 of 1] Compiling Main
Ok, modules loaded: Main .
*Main> isDigit 'g'
False
*Main> isDigit '8'
True
*Main> even 44
*Main> estePar 56
True
*Main> rupeLa 3 [1,2,3,4,5,6]
([1,2,3],[4,5,6])
```



Exemple

- Pot fi definite funcții și "on-line":

```
Prelude> let semiP a b c = (a+b+c)/2
Prelude> let ariaT p a b c = sqrt (p*(p-a)*(p-b)*(p-c))
Prelude> ariaT (semiP 3 4 5) 3 4 5
6.0
```

```
Prelude> let x1 a b c = (-b+sqrt(b*b-4*a*c))/(2*a)
Prelude> let x2 a b c = (-b-sqrt(b*b-4*a*c))/(2*a)
Prelude> x1 1 5 6
-2.0
Prelude> x2 1 5 6
-3.0
```

```
Prelude> let radacini a b c = let {disc = sqrt(b*b-4*a*c);doia =
    2*a} in ((-b+disc)/doia, (-b-disc)/doia)
Prelude> radacini 1 2 1
(-1.0,-1.0)
Prelude> radacini 1 12 35
(-5.0,-7.0)
```



Definirea funcțiilor

- Expresii condiționale:

if condiție then res1 else res2

- *res1* și *res2* sunt expresii de același tip
- Nu există **if** fără **else** în Haskell

semn :: Int -> Int

```
semn n = if n < 0 then -1 else  
          if n == 0 then 0 else 1
```



Definirea funcțiilor

- Expresii case:

```
f1 x =  
  case x of  
    0 -> 1  
    1 -> x + 2  
    2 -> x * 3  
    _ -> -1  
  
f2 x = case x of  
  {0 -> 10; 1 -> 2*x; 2 -> x*x+2; _ -> 0}
```

Case> f1 1

3

Case> f2 2

6



Definirea funcțiilor

```
Prelude> let mySign x = if x < 0 then -1 else if x == 0 then 0
      else 1
Prelude> mySign 9
1
Prelude> mySign (-99)
-1
Prelude> mySign 0
0

Prelude> let f x = case x of { 1 -> 1;2 -> x * x;3 -> x+x;4 -> 0;_
      -> 10}
Prelude> f 5
10
Prelude> f 3
6
Prelude> f 2
4
Prelude> f 0
10
```



Definirea funcțiilor

- Ecuății gardate (cu gărzi)

- Alternativă la condiții cu if
 - Secvență de expresii logice: gărzi
 - Secvență de expresii (rezultate) de același tip
 - Sintaxa:

nume_func parametri | *garda1* = *exp1*
 | *garda2* = *exp2*
 ...
 | *gardan* = *expn*

- Ultima gardă poate fi **otherwise** (definită în bibliotecă cu valoarea **True**)



Exemple

```
semng n      | n < 0      = -1
               | n == 0     = 0
               | otherwise = 1
```

```
*Main> semng 7
```

```
<interactive>:1:0: Not in scope: `semng'
```

```
*Main> :r
[1 of 1] Compiling Main          ( c2test.hs, interpreted )
Ok, modules loaded: Main.
```

```
*Main> semng 7
```

```
1
```

```
*Main> semng -97
```

```
<interactive>:1:0:
```

```
  No instance for (Num (a -> a1))
```

```
    arising from a use of '-' at <interactive>:1:0-8
```

```
  Possible fix: add an instance declaration for (Num  
(a -> a1))
```

```
    In the expression: semng - 97
```

```
    In the definition of `it': it = semng - 97
```

```
*Main> semng (-97)
```

```
-1
```



Definiții locale

- O funcție care are în expresia sa o frază de forma “where ...” spunem că are o definiție locală; definiția din fraza where este valabilă doar local
- Exemplu: $f(x,y) = (a+1)(a+2)$, unde $a = (x+2)/3$

```
f :: (Float, Float) ->Float
f(x,y) = (a+1)*(a+2) where a = (x+2)/3
```

```
f :: (Float, Float) ->Float
f(x,y) = (a+1)*(b+2)
          where a = (x+y)/3
                b = (x+y)/2
```



Definiții locale

```
g  :: Integer -> Integer -> Integer
g x y      | x <= 10 = x + a
              | x > 10 = x - a
where a = square(y+1)
```

- Clauza **where** califică ambele ecuații gardate

```
*Main> g 3 4
28
*Main> f (5,6)
35.0
*Main> g 11 2
2
```



```
Prelude> let aria_triunghi a b c =  
    sqrt(p*(p-a)*(p-b)*(p-c)) where p =  
    (a+b+c)/2
```

```
Prelude> aria_triunghi 3 4 5  
6.0
```

```
Prelude> aria_triunghi 35 35 35  
530.4405598179686
```



Definirea funcțiilor

- Tehnica şabloanelor (“pattern matching”)
 - Secvență de şabloane – rezultate
 - Sintaxa:

numefunc :: tipul_funcției

Şablon1 = exp1

Şablon2 = exp2

...

Şablonn = expn

- Dacă se potrivește primul şablon, se alege ca rezultat *exp1*, dacă nu se încearcă al doilea şablon, etc.
- Şablonul _ (“wildcard”) poate fi folosit pentru a exprima potrivirea cu orice valoare



Exemple

non :: Bool -> Bool

non False = True

non True = False

conj :: Bool -> Bool -> Bool

conj True True = True

conj True False = False

conj False True = False

conj False False = False

(&) :: Bool -> Bool -> Bool

True & True = True

_ & _ = False



Exemple

```
*Main> :type non
non :: Bool -> Bool
*Main> :type conj
conj :: Bool -> Bool -> Bool
*Main> conj (1<2) (2<11)
True
*Main> (1<2) `conj` (2<11)
True
*Main> (7-2>5) & (2<9)
False
*Main> (7-2>=5) & (2<9)
True
```



Exemplu

- O altă modalitate de utilizare a şabloanelor (utilă în evaluarea lazy): dacă primul argument este True (la conjuncție) rezultatul este valoarea celui de-al doilea

(&&) :: Bool -> Bool -> Bool

True && b = b

False && _ = False

(||) :: Bool -> Bool -> Bool

False || b = b

True || _ = True



Exemplu

- Nu se poate folosi numele unui argument de două ori:

```
(&&) :: Bool -> Bool -> Bool
```

```
b && b = b
```

```
False && _ = False
```

- Soluția: folosirea gărzilor:

```
(&) :: Bool -> Bool -> Bool
```

```
b & c | b == c = b
```

```
    | otherwise = False
```



Definirea funcților

- Tehnica şablonelor (“pattern matching”):
 - Şabloane de tip tuple: o tuplă de şabloane este un şablon

```
fst :: (a, b) -> a
```

```
fst (x, _) = x
```

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

```
snd (_, y) = y
```



Definirea funcțiilor

- Tehnica şabloanelor (“pattern matching”):
 - Şabloane de tip listă: o listă de şabloane este un şablon

```
test3a    :: [Char] -> Bool
```

```
test3a ['a', _, _] = True
```

```
test3a _ = False
```

```
*Main> test3a ['a', 'c', 'r']
```

```
True
```

```
*Main> test3a ['a', 'c', 'r', 'a']
```

```
False
```



Utilizarea operatorului ":" la liste (cons)

- Orice listă este construită prin repetarea operatorului cons(":") care înseamnă adăugarea unui element la o listă
 - Lista [1,2,3,4] înseamnă
$$1:(2:(3:(4:[])))$$

```
test      :: [Char] -> Bool
test ('a': _) = True
test _ = False
• Şabloanele listă trebuie puse în paranteză
```



```
*Main> :type test
test :: [Char] -> Bool
*Main> test ['q', 'w', 'e']
False
*Main> test ['a', 'q', 'w', 'e']
True
```

```
null :: [a] -> Bool
nul [] = True
nul (_:_ ) = False
```

```
head :: [a] -> a
head (x:_ ) = x
```

```
tail :: [a] -> [a]
tail (_:xs) = xs
```



Definirea funcțiilor

- Tehnica şabloanelor (“pattern matching”):
 - Şabloane “întregi”: expresii de forma $n+k$ unde n este o variabilă întreagă iar k este o constantă întreagă pozitivă

```
pred      :: Int -> Int
```

```
pred 0      = 0
```

```
pred(n+1)  = n
```

- Şabloanele întregi se potrivesc doar cu expresii pozitive
- Şabloanele întregi trebuie puse în paranteză



Cursul 3-4 – Plan

- Lambda calcul
 - Lambda calcul – formal
 - Teoria ecuațiilor în Λ , Beta reducere
 - Forme normale, reprezentări
- Lambda expresii în Haskell
- Definiții recursive
- Definiții locale
- Liste, Operații cu liste



Scurtă istorie

- λ -calculus: introdus de **Alonzo Church** (14.06.1903 – 11.08.1995) în 1936 (1940 - λ -calculus cu tipuri) – proiectul “Fundamentele matematicii”
- Sistem formal pentru definirea funcțiilor, aplicarea funcțiilor și recursie
- Ideile au condus la aplicații în logică, teoria calculabilității, lingvistică, teoria limbajelor de programare (programare funcțională, sistemul de tipuri)



Lambda calcul – scurtă introducere

Lambda calculul se bazează pe două operații:

- **Aplicația:**
 - F.A care se scrie de obicei FA
 - F privit ca algoritm, A considerat ca intrare
 - În particular FF (recursie)
- **Abstracția:**
 - Dacă $M = M[x]$ este o expresie, atunci funcția $x \rightarrow M[x]$ se notează $\lambda x. M[x]$



Lambda calcul formal

- Fie V o mulțime infinită de variabile
- Sintaxa (BNF) **lambda expresiilor** (Exp):

Exp ::= Var | Exp Exp | λ Var.Exp

unde Var are valori din V

- Exemple:
 $\lambda x.x, \lambda x.xx, \lambda x.(fx)(gx), (\lambda x.fx)x$
- Notații:
 - $FM_1M_2\dots M_n$ notează $(\dots((FM_1)M_2)\dots M_n)$
 - $\lambda x_1x_2\dots x_n.M$ notează $\lambda x_1.(\lambda x_2.(\dots (\lambda x_n. (M))\dots))$



Lambda calcul

- Variabile
 - **libere**: în $\lambda x.yx$ variabila y este liberă
 - **legate**: în $\lambda x.yx$ variabila x este legată
- Substituție: $M[x:=N]$ aparițiile libere ale lui x se înlocuiesc cu N
 - $yx(\lambda x.x)[x:=N] = yN(\lambda x.x)$
- $(\lambda x.M[x])N = M[x:=N]$
 - $(\lambda x.2*x+1)3 = 2*3 + 1$



Lambda calcul - formal

- Multimea variabilelor libere ale lui M, $FV(M)$, este:
 - $FV(x) = \{x\}$
 - $FV(MN) = FV(M) \cup FV(N)$
 - $FV(\lambda x.M) = FV(M) - \{x\}$
- Substituție: $M[x:=N]$ este M în care orice apariție liberă a lui x se înlocuiește cu N. Formal:
 - $y[x := E'] = \text{if } y = x \text{ then } E' \text{ else } y$
 - $(E_1 E_2)[x := E'] = (E_1[x := E'])(E_2[x := E'])$
 - $(\lambda x.E)[x := E'] = \lambda x.E$
 - $(\lambda y.E)[x := E'] =$
 $= \lambda y.(E[x := E']) \quad \text{if } y \notin FV(E')$
 $= \lambda z.((E[y := z])[x := E']) \quad \text{if } y \in FV(E')$



Lambda calcul - formal

- M este term închis, sau **combinator**, dacă $FV(M) = \emptyset$. Multimea combinatorilor se notează Λ^0
- Combinatorii standard:
 - $I = \lambda x.x$ $K = \lambda xy.x$ $S = \lambda xyz.xz(yz)$
 - $IM = M$ $KMN = M$ $SMNL = ML(NL)$



Teoria ecuațiilor în Λ

- Axiome

(β) $(\lambda x.M)N = M[x:=N]$ pentru orice $M, N \in \Lambda$

$$M = M$$

$$M = N \Rightarrow N = M; M = N, N = L \Rightarrow M = L$$

$$M = M' \Rightarrow MZ = M'Z, ZM = ZM'$$

(ξ) $M = M' \Rightarrow \lambda x.M = \lambda x.M'$

(α) $\lambda x.M = \lambda y.M[x := y]$

- Dacă se poate demonstra că $M = N$, spunem ca M și N sunt β -convertibili
- Notăm $M \equiv N$ dacă M și N sunt aceeași modulo redenumirea variabilelor legate: $(\lambda x.x)z \equiv (\lambda y.y)z$



Beta reducere

- Motorul de calcul al sistemului
- Un redex este o subexpresie de forma:
 $(\lambda x.E1)E2$
- Prin reducere se obține $E1[x:=E2]$: orice apariție liberă a variabilei x în $E1$ se substituie cu $E2$



Exemplul 1

$(\lambda x.yxzx(\lambda x.yx)x)(abc)$ (redex?)

$(\lambda x.yxzx(\lambda x.yx)x)(abc) \rightarrow$

$(yxzx(\lambda x.yx)x)[x:=abc]$ (apariții libere x?)

$(y\cancel{x}z\cancel{x}(\lambda x.yx)\cancel{x})[x:=abc] \rightarrow$

$y(\cancel{abc})z(\cancel{abc})(\lambda x.yx)(\cancel{abc})$



Exemplul 2

$(\lambda f g x. f(gx))(\lambda a. a)(\lambda b. bb)c \rightarrow$

$(\lambda g x. (\lambda a. a) (gx))(\lambda b. bb)c \rightarrow$

$(\lambda g x. gx)(\lambda b. bb)c \rightarrow$

$(\lambda x. (\lambda b. bb)x)c \rightarrow$

$(\lambda x. xx)c \rightarrow$

CC



Exemplul 3

$$(\lambda x.a.x)a \rightarrow \lambda a.(\lambda x.x)a \rightarrow \lambda a.aa$$

În $\lambda a.(\lambda x.x)a$, **a** a devenit legat și era liber!
 $(\lambda x.x)a$

Corect: (se aplică o reducere alfa)

$$\lambda x.a.x = \lambda x.b.x$$

$$(\lambda x.b.x)(\lambda x.x)a \rightarrow \lambda b.(\lambda x.x)a \rightarrow \lambda b.ba$$



Forme normale

- Un term fără redex-uri este o **formă normală**
- Formele normale corespund rezultatului calculului (valori în Haskell)
- Nu toți termii au forme normale:
 $(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow \dots$
 $(\lambda x.fff)(\lambda x.fff) \rightarrow (\lambda x.fff)(\lambda x.fff)(\lambda x.fff) \rightarrow \dots$



Forme normale

- Pot exista mai multe strategii de reducere:

$$(\lambda f g x. \textcolor{red}{f(gx)}) (\lambda a. a) (\lambda b. bb) c \rightarrow$$
$$(\textcolor{blue}{\lambda} g x. \underline{(\lambda a. a)(gx)}) (\lambda b. bb) c \rightarrow ?$$
$$(\lambda f. (\lambda x. f x) y) g \rightarrow (\lambda f. f y) g$$
$$(\lambda f. (\lambda x. f x) y) g \rightarrow (\lambda x. g x) y$$



Forme normale

- **Teorema confluenței:** Relația \rightarrow este confluentă: dacă $E \rightarrow^* E_1$ și $E \rightarrow^* E_2$ atunci există E' astfel ca $E_1 \rightarrow^* E'$ și $E_2 \rightarrow^* E'$
(Toate strategiile conduc la aceeași formă normală)
- Nu toate strategiile sunt bune pentru a termina calculul
- Dacă este posibil(calculul se termină), există o strategie care conduce la forma normală: **call by name reduction**: se alege totdeauna redexul cel mai din stânga al termului
- Această strategie este cunoscută ca și **lazy evaluation** (Haskell)



Reprezentarea recursiei în lambda calcul

“To understand recursion, one must first understand recursion.”

Teorema de punct fix

- (1) $\forall F \in \Lambda, \exists X \in \Lambda$ astfel încât $FX = X$
- (2) Există combinatorul de punct fix

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

astfel încât $\forall F \in \Lambda F(YF) = YF$

Demonstratie

- (1) Fie $W \equiv \lambda x.F(xx)$ și $X \equiv WW$ Atunci
 $X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv F(X)$
- (2) La fel



Exemplu

- Fie:

$$F = \lambda f. \lambda x. (\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x - 1))$$

$$(Y F) 3 =$$

$$F (Y F) 3 =$$

$$(\lambda f. \lambda x. (\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x - 1))) (Y F) 3$$

$$\text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (Y F)(3 - 1)$$

$$3 * ((Y F) 2)$$

...

$$6 * ((Y F) 0)$$

$$6 * (F (Y F) 0) =$$

$$6 * ((\lambda f. \lambda x. (\text{if } x == 0 \text{ then } 1 \text{ else } x * f(x - 1))) (Y F) 0)$$

$$6 * \text{if } 0 == 0 \text{ then } 1 \text{ else } 0 * (Y F)(0 - 1)$$

$$6 * 1$$

$$6$$



Reprezentarea numerelor și operațiilor

Definiție. $F^0(M) = M$, $F^{n+1}(M) = F(F^n(M))$

Numerale Church: $c_0, c_1, \dots, c_n \equiv \lambda f x. f^n x$

Operații: $A_+ \equiv \lambda x y p q. x p(y p q)$

$A_* \equiv \lambda x y z. x(y z)$

$A_{\text{exp}} \equiv \lambda x y. y x$

Teoremă. Pentru orice număr natural n, m :

$$(1) A_+ c_n c_m = c_{n+m}$$

$$(2) A_* c_n c_m = c_{n * n}$$

$$(3) A_{\text{exp}} c_n c_m = c_{n^m} \text{ (n diferit de zero)}$$



Reprezentarea funcțiilor

Definiție.

$\text{true} \equiv K \equiv \lambda xy.x, \text{ false} \equiv K_* \equiv \lambda xy.y$

$\text{if-then-else} \equiv \lambda xyz.xyz$

$\text{if } B \text{ then } P \text{ else } Q$ poate fi reprezentat prin $B P Q$

$\text{and} \equiv \lambda xy.(x \text{ y} \text{ false})$

Perechea ordonată: $[M, N] \equiv \lambda z.MN$

Numerale: $\underline{0} \equiv I \equiv \lambda x.x, \underline{n+1} \equiv [\text{false}, \underline{n}]$



Reprezentarea funcțiilor

Lemă Există combinatorii S^+ , P^- , zero astfel ca pentru orice număr natural n au loc: $S^+ \underline{n} = \underline{n+1}$, $P^- \underline{n+1} = \underline{n}$, Zero $\underline{0} = \text{true}$, Zero $\underline{n+1} = \text{false}$

Demonstratie:

$$S^+ \equiv \lambda x. [\text{false}, x]$$

$$P^- \equiv \lambda x. x \text{ false}$$

$$\text{Zero} \equiv \lambda x. x \text{ true}$$



Reprezentarea funcțiilor

Definitie. O funcție numerică $f : N^p \rightarrow N$ este λ -definibilă dacă există un combinator F astfel ca

$$F \underline{n}_1 \underline{n}_2 \dots \underline{n}_p = f(\underline{n}_1, \underline{n}_2, \dots, \underline{n}_p)$$

Teoremă Funcțiile recursive sunt λ -definibile:

- (1) Funcțiile inițiale(zero, succesor, proiecția) sunt λ -definibile
- (2) Clasa funcțiilor λ -definibile este închisă la compunere($f(n) = g(h_1(n), \dots, h_m(n))$)
- (3) Clasa funcțiilor λ -definibile este închisă la recursie primitivă ($f(0, n) = g(n)$, $f(k+1, n) = h(f(k, n), k, n)$)
- (4) Clasa funcțiilor λ -definibile este închisă la minimizare
($f(n) = \mu m [g(n, m) = 0]$)



Lambda expresii în Haskell

- Sintaxa:

\x -> exp

- Expresia **exp** conține x; poate fi o nouă lambda expresie
- Exemple de funcții definite ca lambda expresii:

\x -> x + x

\x -> (\y -> x + y)

\x -> (\y -> x*x + y*y)

\x y -> (x*x + y*y)



Exemple

```
Prelude> (\x->x+x) 8
16

Prelude> (\x y-> x*x+y*y) 3 4
25

Prelude> (\x -> sum[1..x]) 5
15

Prelude> let const x = \_ -> x
Prelude> const 8 5
8

Prelude> let const1 x = \y -> x
Prelude> const1 8 5
8

Prelude> const1 [1,2,3] 5
[1,2,3]

Prelude> const [1,2,3] 5
[1,2,3]

Prelude> const [1,2,3] (0,9)
[1,2,3]

Prelude> const False [1,2,3]
False
```



Exemplu

- Folosim funcția `map` pentru a obține primele n numere impare:

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

```
odds :: Int -> [Int]
```

```
odds n = map f [0..n-1]
```

```
where f x = x*x + 1
```

- Alternativă (lambda expresie):

```
odds n = map (\x -> x*x + 1) [0..n-1]
```



Secțiuni

- Funcțiile cu două argumente pot fi utilizate infix(ca operatori): $12 \text{ `div' } 3$
- Operatorii (+, *, etc.) pot fi utilizați ca funcții:
 $(+)$ 3 4 $(3+)$ 4 $(+4)$ 3
- Dacă @ este un operator atunci expresiile de forma $(@)$, $(x@)$, $(@y)$ se numesc secțiuni și sunt funcții:
 $(@) = \lambda x \rightarrow (\lambda y \rightarrow x @ y)$
 $(x @) = \lambda y \rightarrow x @ y$
 $(@ y) = \lambda x \rightarrow x @ y$



Exemple

```
Prelude> :t (+)
(+) :: (Num a) => a -> a -> a
Prelude> (+) 4 5
9
Prelude> :t (4+)
(4+) :: (Num a) => a -> a
Prelude> (4+) 5
9
Prelude> :t (+5)
(+5) :: (Num a) => a -> a
Prelude> (+5) 4
9

Prelude> :t flip
flip :: (a -> b -> c) -> b -> a -> c
Prelude> (-) 3 9
-6
Prelude> flip (-) 3 9
6
```



Aplicații ale secțiunilor

- Construirea de funcții simple

$(1+) = \lambda y \rightarrow 1+y$

$(1/) = \lambda y \rightarrow 1/y$

$(*2) = \lambda x \rightarrow x^2$

$(/2) = \lambda x \rightarrow x/2$

- Aflarea tipului operatorilor
- Utilizarea operatorilor ca argumente pentru alte funcții



Definiții recursive

- O funcție care are în expresia ce definește valoarea sa numele său este o funcție recursivă

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

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



Definiții recursive

```
fact(-1) = (definitie)
if -1 == 0 then 1 else (-1)*fact(-1-1)=
(-1) * fact(-1-1) =
(-1)*if (-1-1) == 0 then 1 else (-1-1)*fact((-1-1)-1)=
(-1)*((-2)*fact(-1-1-1))
...
...
```

- Redefinim funcția:

```
fact      :: Integer -> Integer
fact n    | n<0 = error "argument negativ"
           | n == 0 = 1
           | n>0  = n*fact(n-1)
```

```
error :: String -> a
```



Definiții recursive

```
Prelude> let fact n = if n < 0 then error "argument negativ"  
    else if n == 0 then 1 else n*fact (n-1)
```

```
Prelude> fact 33  
8683317618811886495518194401280000000  
Prelude> fact (-4)  
*** Exception: argument negativ
```

```
Prelude> :t fact  
fact :: (Num a, Ord a) => a -> a  
Prelude> fact 3.5  
*** Exception: argument negativ  
Prelude> fact 3.999999999999999  
*** Exception: argument negativ  
Prelude> fact 3.999999999999999  
24.0
```



Definiții recursive

- O funcție recursivă se definește astfel:

- Se definește tipul:

```
produs :: [Int] -> Int
```

- Se enumeră cazurile:

```
produs []      =
```

```
produs (n:ns) =
```

- Se definesc cazurile de bază:

```
produs []      = 1
```

```
produs (n:ns) =
```

- Se definesc celelalte cazuri

```
produs []      = 1
```

```
produs (n:ns) = n * produs ns
```

- Generalizări, simplificări:

```
produs :: Num a => [a] -> a
```



Stil declarativ vs. Stil expresie

- Există două stiluri de a scrie programe funcționale:
 - Stilul declarativ: definirea funcțiilor prin ecuații multiple, fiecare din acestea utilizând “pattern matching” și/sau gărzi pentru a identifica toate cazurile
 - Stilul expresie: compunerea de expresii pentru a obține alte expresii



Stil declarativ vs. Stil expresie

```
factorial :: Int -> Int
```

- Stilul declarativ:

```
factorial n = iter n 1
where
    iter 0 r = r
    iter n r = iter (n-1) (n*r)
```

- Stilul expresie:

```
factorial n =
let iter n r =
    if n == 0 then r
    else iter (n-1) (n*r)
in iter n 1
```



Stil declarativ vs. Stil expresie

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

- Stilul declarativ:

```
filter p [] = []
```

```
filter p (x:xs)  | p x = x : rest  
                   | otherwise = rest  
where
```

```
rest = filter p xs
```



Stil declarativ vs. Stil expresie

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

- Stilul expresie:

```
filter = \p -> \xs ->
          case xs of
            [] -> []
            (x:xs) -> let
                          rest = filter p xs
                          in if (p x)
                             then x : rest
                             else rest
```



Caracteristici sintactice

- Stilul declarativ:
 - Folosește clauze where
 - Argumentele funcției sunt în partea stângă
 - Folosește pattern matching
 - Folosește gărzi în definirea funcțiilor
- Stilul expresie
 - Folosește expresii let
 - Folosește lambda abstracții
 - Folosește expresii case
 - Folosește expresii if



Liste

- Tipul listă

[1,2,3,4] :: Num a => [a]

[[1,2,3,4],[1,2]] :: Num a => [[a]]

[(+),(*),(-)] :: Num a => [a -> a -> a]

- Constructorul (cons) :

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

[1,2,3] = 1:(2:(3:[])) = 1:2:3:[]

- Funcția null

null :: [a] -> Bool

null [] = True

null x:xs = False

Hugs.Base> null []

True

Hugs.Base> null [1,2]

False



Operații cu liste

- Concatenare: ++

$(++) :: [a] \rightarrow [a] \rightarrow [a]$

$[] ++ ys = ys$

$(x:xs) ++ ys = x:(xs++ys)$

$[1,2] ++ [3,4,5] = \{notatie\}$

$(1:(2:[])) ++ (3:(4:(5:[]))) = \{def\ ++, ec2\}$

$1:((2:[]) ++ (3:(4:(5:[])))) = \{def\ ++, ec2\}$

$1:(2:([] ++ (3:(4:(5:[]))))) = \{def\ ++, ec1\}$

$1:(2:(3:(4:(5:[])))) = \{notatie\}$

$[1,2,3,4,5]$



Operații cu liste

- Concatenarea (++) este asociativă iar [] este element neutru:

$$(xs \text{ ++ } ys) \text{ ++ } zs = xs \text{ ++ } (ys \text{ ++ } zs)$$
$$xs \text{ ++ } [] = [] \text{ ++ } xs$$

- Funcția **concat** concatenează o listă de liste:

$$\text{concat} :: [[a]] \rightarrow [a]$$
$$\text{concat} [] = []$$
$$\text{concat}(xs : xss) = xs \text{ ++ concat } xss$$

```
Prelude> concat [[1,2,3], [4], [], [5,6]]  
[1,2,3,4,5,6]
```



Funcția **reverse**

- Funcția **reverse** inversează elementele unei liste:

```
reverse :: [a] -> [a]
```

```
reverse[] = []
```

```
reverse (x:xs) = reverse xs ++ [x]
```

- Funcția **reverse** realizează inversarea unei liste de lungime n în $n(n-1)$ pași

```
Prelude > reverse [1,2,3,4,5,6]
```

```
[6,5,4,3,2,1]
```

```
Prelude > reverse (reverse [1,2,3,4,5,6])
```

```
[1,2,3,4,5,6]
```

```
Prelude > reverse "epurasulusarupe"
```

```
"epurasulusarupe"
```



Funcția `length`

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

- Se poate dovedi (de la definițiile ++ și `length`):
`length (xs++ys) = length xs + length ys`
- Funcția `length` calculează lungimea unei liste în n pași indiferent de natura elementelor

```
Prelude > length [1,2]
```

```
2
```

```
Prelude > length [undefined, undefined]
```

```
2
```



Funcțiile **head**, **tail**

head :: [a] -> a

head[x:xs] = x

tail :: [a] -> [a]

tail(x:xs) = xs

- Sunt operații ce se execută în timp constant
- Componere de funcții (.) :
 $(.) :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$

$$(f . g) x = f(gx)$$



Functiile last, init

```
last :: [a] -> a
```

```
last = head.reverse
```

```
Prelude > (head.reverse) [1,2,3]
```

```
3
```

```
Prelude > last[1,2,3]
```

```
3
```

```
init :: [a] -> [a]
```

```
init = reverse.tail.reverse
```

```
Prelude > init [1,2,3]
```

```
[1,2]
```

```
Prelude > (reverse.tail.reverse) [1,2,3]
```

```
[1,2]
```



Functia take

- Functia **take**: primele n elemente din lista xs
- Definiție recursivă:
 - Tipul funcției:

```
take    :: Int -> [a] -> [a]
```
 - Enumerarea cazurilor: sunt 2 valori pentru primul argument: 0 și n+1, două pentru al doilea argument: [] și x:xs

<code>take 0 []</code>	=
<code>take 0 (x:xs)</code>	=
<code>take (n+1) []</code>	=
<code>take (n+1) (x:xs)</code>	=

- Definirea cazurilor de bază:

<code>take 0 []</code>	= []
<code>take 0 (x:xs)</code>	= []
<code>take (n+1) []</code>	= []
<code>take (n+1) (x:xs)</code>	=



Funcția `take`

- Definirea celorlalte cazuri:

```
take 0 []          = []
take 0 (x:xs)      = []
take (n+1) []       = []
take (n+1) (x:xs)  = x:take n xs
```

- Generalizare, simplificare:

```
take :: Integral b => b -> [a] -> [a]
```

```
take 0 xs          = []
take (n+1) []       = []
take (n+1) (x:xs)  = x:take n xs
```



Functiile **take**, **drop**

```
take      :: Int -> [a] -> [a]
take 0 xs = []
take (n+1) [] = []
take (n+1) (x:xs) = x:take n xs
```

```
drop      :: Int -> [a] -> [a]
drop 0 xs = xs
drop (n+1) [] = []
drop (n+1) (x:xs) = drop n xs
```



Funcțiile `take`, `drop`

- Proprietăți:

<code>take n xs ++ drop n xs</code>	$= \text{xs}$
<code>take m . take n</code>	$= \text{take}(\text{m} \text{ `min' } \text{n})$
<code>drop m . drop n</code>	$= \text{drop}(\text{m+n})$
<code>take m . drop n</code>	$= \text{drop n. take}(\text{m+n})$

- Exercițiu: Demostrați relațiile de mai sus pornind de la definiții

```
Prelude > ((take 3) . (take 5)) [1] == take(3`min`5) [1]
```

`True`

```
Prelude > (drop 2. drop 4) [1] == drop(2+4) [1]
```

`True`

```
Prelude > (take 5.drop 3) [1] == (drop 3.take(5+3)) [1]
```

`True`



Funcția splitAt

```
splitAt :: Int -> [a] -> ([a], [a])
```

```
splitAt n xs = (take n xs, drop n xs)
```

```
splitAt 0 xs          = ([] , xs)
```

```
splitAt n+1 []        = ([] , [])
```

```
splitAt n+1 (x:xs)    = (x:ys, zs)
```

```
where (ys,zs)=splitAt n xs
```



Indexarea în liste

- Operatorul de indexare (!!) :

```
(!!) :: [a] -> Int -> a
```

```
(x:xs) !! 0           = x
(x:xs) !! (n+1)      = xs !! n
```

- Proprietăți:

```
(xs ++ ys) !! k = if k < n then xs !! k else ys !! (k-n)
                    where n = length xs
```

```
Prelude > "alabalaportocala" !! 12
```

```
'c'
```

```
Prelude > "aaaa" !! (-1)
```

*****Exception: Prelude. (!!): negative index**

```
Prelude > ("aaa" ++ "bbb") !! 4
```

```
'b'
```



Funcția map

- Funcția map aplică o funcție fiecărui element al unei liste

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

```
map f [] = []
```

```
map f(x:xs) = fx : map f xs
```

```
Prelude > map square [2,1,4]
```

```
[4,1,16]
```

```
Prelude > map (<5) [ 3,7,5,4,3,7,8]
```

```
[True, False, False, True, True, False, False]
```

```
Prelude > sum(map square[1..3])
```

```
14
```

```
Prelude > sum(map sqrt[1..8])
```

```
16.3060005260357
```



Funcția map

- Proprietăți:

`map id`

`= id`

`map(f.g)`

`= map f.map g`

`f.head`

`= head . map f`

`map f.tail`

`= tail.map f`

`map f.reverse`

`= reverse.map f`

`map f.concat`

`= concat.map(map f)`

`map f (xs++ys) = map f xs ++ map f ys`



Funcția filter

- Argumente: o funcție booleană p și o listă xs
- Returnează sublista elementelor din xs ce satisfac p

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)= if p x then
                  x:filter p xs
                else filter p xs
```

```
Prelude > filter odd [1,2,3,4,5,6,7,8]
```

```
[1,3,5,7]
```

```
Prelude > (sum.map (*4).filter even) [1..10]
```

```
120
```



Funcția filter

- Proprietăți:

```
filter p . filter q = filter(p `si` q)
unde (p `si` q)x = px && qx
filter p . concat = concat . map(filter p)
```

```
Prelude > (filter (>3).filter(<7))[1,2,3,4,5,6,7,8,9]
[4,5,6]
```

```
Prelude > (filter (<5).concat)[[1,3,6],[3,8]]
[1,3,3]
```

```
Prelude > (concat.map(filter(<5)))[[1,3,6],[3,8]]
[1,3,3]
```



Cursul 4 - Plan

- Operații cu liste (cont)
- O altă reprezentare a listelor (**comprehension**)
 - Generatori
 - Gărzi
- zip, unzip, cross
- foldr, foldl
- scanl, scanr
- Studii de caz



Indexarea în liste

- Operatorul de indexare (!!) :

```
(!!) :: [a] -> Int -> a
```

```
(x:xs) !! 0           = x
(x:xs) !! (n+1)      = xs !! n
```

- Proprietăți:

```
(xs ++ ys) !! k = if k < n then xs !! k else ys !! (k-n)
                    where n = length xs
```

```
Prelude > "alabalaportocala" !! 12
```

```
'c'
```

```
Prelude > "aaaa" !! (-1)
```

*****Exception: Prelude. (!!): negative index**

```
Prelude > ("aaa" ++ "bbb") !! 4
```

```
'b'
```



Funcția map

- Funcția map aplică o funcție fiecărui element al unei liste

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

```
map f [] = []
```

```
map f(x:xs) = fx : map f xs
```

```
Prelude > map square [2,1,4]
```

```
[4,1,16]
```

```
Prelude > map (<5) [ 3,7,5,4,3,7,8]
```

```
[True, False, False, True, True, False, False]
```

```
Prelude > sum(map square[1..3])
```

```
14
```

```
Prelude > sum(map sqrt[1..8])
```

```
16.3060005260357
```



Funcția map

- Proprietăți:

`map id`

`= id`

`map(f.g)`

`= map f.map g`

`f.head`

`= head . map f`

`map f.tail`

`= tail.map f`

`map f.reverse`

`= reverse.map f`

`map f.concat`

`= concat.map(map f)`

`map f (xs++ys) = map f xs ++ map f ys`



Funcția filter

- Argumente: o funcție booleană p și o listă xs
- Returnează sublista elementelor din xs ce satisfac p

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)= if p x then
                  x:filter p xs
                else filter p xs
```

```
Prelude > filter odd [1,2,3,4,5,6,7,8]
```

```
[1,3,5,7]
```

```
Prelude > (sum.map (*4).filter even) [1..10]
```

```
120
```



Funcția filter

- Proprietăți:

```
filter p . filter q = filter(p `si` q)
unde (p `si` q)x = px && qx
filter p . concat = concat . map(filter p)
```

```
Prelude > (filter (>3).filter(<7))[1,2,3,4,5,6,7,8,9]
[4,5,6]
```

```
Prelude > (filter (<5).concat)[[1,3,6],[3,8]]
[1,3,3]
```

```
Prelude > (concat.map(filter(<5)))[[1,3,6],[3,8]]
[1,3,3]
```



Liste: o altă notație

- Analogie cu definirea unei mulțimi:
 $\{1, 4, 9, 16, 25\} = \{x^2 \mid x \in \{1..5\}\}$
- “List **comprehension**”:

```
Prelude > [x*x|x<-[1..5]]
```

```
[1,4,9,16,25]
```

```
Prelude > [x*x|x<-[1..5], odd x]
```

```
[1,9,25]
```

```
Prelude > [x*x|x<-[1..10], even x]
```

```
[4,16,36,64,100]
```



Liste – “comprehension”

- Sintaxa Haskel pentru definirea listelor:

$[e \mid Q]$ unde

- e este o expresie
- Q este un calificator
 - O secvență – posibil vidă – de forma
gen1, gen2, ...
 - Generator: **x <- xs**
 - x este variabilă sau tuplă de variabile
 - xs este o expresie cu valori liste
 - Gardă: o expresie cu valori booleene (gărzile pot lipsi)



Liste

- Dacă în expresia $[e \mid Q]$ calibratorul Q este vid atunci scriem doar $[e]$

- Regula generator:

```
[e|x<- xs, Q] = concat(map f xs) where fx = [e|Q]
```

- Regula gardă:

```
[e|p, Q] = if p then [e|Q] else []
```



Exemple

```
Prelude > [(x,y) | x<-[1..5], y<-[1..x], x+y<6]
[(1,1), (2,1), (2,2), (3,1), (3,2), (4,1)]
```

```
Prelude > [(x,y) | x<-[0..9], x<=3, y<-[2..4]]
[(0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,2), (3,3), (3,4)]
```

```
Prelude > [x| x <- "Facultatea de Informatica", 'c'<= x && x <= 'h']
"cedefc"
```

```
Prelude > [(x,y,z) | x<-[0..5], y<-[0..5], z<-[0..5], x+y+z == 3]
[(0,0,3), (0,1,2), (0,2,1), (0,3,0), (1,0,2), (1,1,1), (1,2,0), (2,0,1),
(2,1,0), (3,0,0)]
```

```
Prelude > [(x,y,z) | x<-[1..5], y<-[x..5], z<-[y..5], x+y > z ]
[(1,1,1), (1,2,2), (1,3,3), (1,4,4), (1,5,5), (2,2,2), (2,2,3), (2,3,3),
(2,3,4), (2,4,4), (2,4,5), (2,5,5), (3,3,3), (3,3,4), (3,3,5), (3,4,4),
(3,4,5), (3,5,5), (4,4,4), (4,4,5), (4,5,5), (5,5,5)]
```



Exemple

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

```
Prelude > quicksort [3,2,5,4,2,6,4,5]
```

```
[2,2,3,4,4,5,5,6]
```

```
Prelude > quicksort ["luni", "marti", "miercuri", "joi",
```

```
  "vineri"]
```

```
["joi", "luni", "marti", "miercuri", "vineri"]
```



Proprietăți

$[f\ x | x \leftarrow xs] = map\ f\ xs$

$[x | x \leftarrow xs, p\ x] = filter\ p\ xs$

$[e | Q, P] = concat[[e | P] | Q]$

$[e | Q, x \leftarrow [d | P]] = [e[x := d] | Q, P]$



Funcția **zip**

- Listele xs și ys sunt transformate într-o listă de perechi cu elemente de pe același loc din cele 2 liste

```
zip    :: [a] -> [b] -> [(a, b)]
zip    [] ys          = []
zip    (x:xs) []       = []
zip    (x:xs) (y:ys)  = (x, y) : (zip xs ys)
```

```
Main> zip [0..4] "hallo"
[(0,'h'),(1,'a'),(2,'l'),(3,'l'),(4,'o')]
Main> zip [1,2,3] "hallo"
[(1,'h'),(2,'a'),(3,'l')]
```



Aplicații ale funcției `zip`

- Produsul scalar:

```
pscalar      :: (Num a) => [a] -> [a] -> a
pscalar xs ys = sum(map times (zip xs ys))
                  where times(x,y) = x*y
```

```
Prelude> pscalar [1, 0, 0] [0, 1, 0]
```

```
0
```

```
Prelude> pscalar [1, 0, 0] [1, 1]
```

```
1
```

- Căutare (pozițiile lui x în lista xs):

```
positions :: (Eq a) => a -> [a] -> [Int]
positions x xs = [i | (i,y) <- zip [0..] xs, x == y]
Prelude > positions 3 [1,2,3,4,3,2,5,3,3,5,4,3]
[2,4,7,8,11]
```



Aplicații ale funcției **zip**

- Funcția **zip xs (tail xs)** returnează lista perechilor de elemente adiacente din xs

```
Hugs> zip [1,2,3,4,5,6,7] (tail [1,2,3,4,5,6,7])
[(1,2), (2,3), (3,4), (4,5), (5,6), (6,7)]
```

- Funcția **zip xs (reverse xs)**

```
Hugs> zip [1,2,3,4,5,6] (reverse [1,2,3,4,5,6])
[(1,6), (2,5), (3,4), (4,3), (5,2), (6,1)]
```



Aplicații ale funcției zip

- este o secvență dată nedescrescătoare?

```
nondec      :: (Ord a) => [a] -> Bool
nondec xs = and (map leq (zip xs (tail xs)))
                 where leq(x,y) = (x <= y)
```

```
Main> nondec [1,2,3,4,5,6,7,8]
```

True

```
Main> nondec [1,2,3,4,5,67,8]
```

False



Funcția unzip

- O listă de perechi $[(x_1, y_1), (x_2, y_2), \dots]$ este transformată într-o pereche de 2 liste $([x_1, x_2, \dots], [y_1, y_2, \dots])$

pair:: (a → b, a → c) → a → (b, c)
pair (f, g) x = (f x, g x)

```
Main> pair ((+2), (*2)) 7  
(9,14)
```

```
Main> pair(and, or) [True, False, False]  
(False,True)
```

```
Main> pair (sum, product) [1,3,5,7]  
(16,105)
```

```
unzip :: [(a, b)] -> ([a], [b])  
unzip = pair (map fst, map snd)
```

```
Main> unzip [(1, 'a'), (2, 'b'), (3, 'c')]  
([1,2,3],"abc")
```



Funcția cross

- Denumirea pentru $f \times g$ din teoria categoriilor:

$$(f \times g)(x, y) = (f(x), g(y))$$

```
cross:: (a -> b, c -> b) -> (a,c) -> (b, b)
cross (f, g) = pair(f.fst, g.snd)
```

```
Main> cross ((+2), (*3)) (10,20)
(12,60)
```

```
Main> cross ((/2), (/3)) (10,20)
(5.0,6.66666666666667)
```

```
Main> cross (and,or) ([True, False, True], [False,
  False, True])
(False,True)
```

```
Main> cross(sum, product) ([10,20,30], [1,2,3])
(60,6)
```



Funcția cross

- Proprietăți

1. **fst.pair(f,g) = f**

```
Main>(fst.pair((<3), (>=3))) 3  
False
```

2. **snd.pair(f,g) = g**

3. **fst.cross(f,g) = f.fst**

4. **snd.cross(f,g) = g.snd**

5. **pair(f,g).h = pair(f.h, g.h)**

6. **cross(f,g).pair(h,k) = pair(f.h, g.k)**

Demonstrație 6:

```
cross(f,g).pair(h,k) = (def cross)
```

```
pair(f.fst, g.snd).pair(h,k) = (prop 5)
```

```
pair(f.fst.pair(h,k), g.snd.pair(h,k)) = (prop 1, 2)
```

```
pair(f.h, g.k)
```



Funcția **foldr** (fold right)

- Utilizată pentru simplificarea definițiilor recursive de forma (@ este o operație):

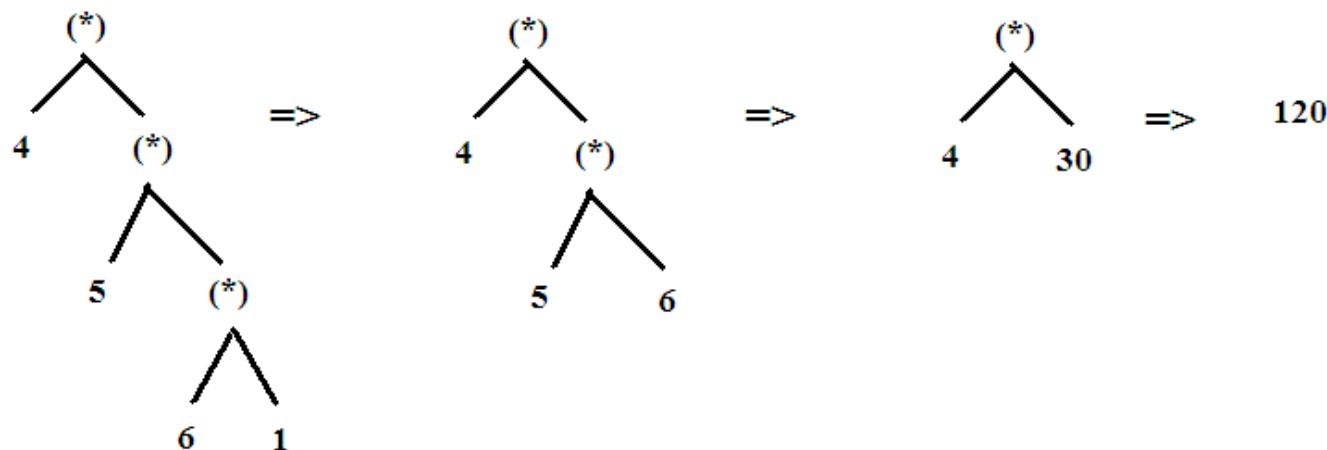
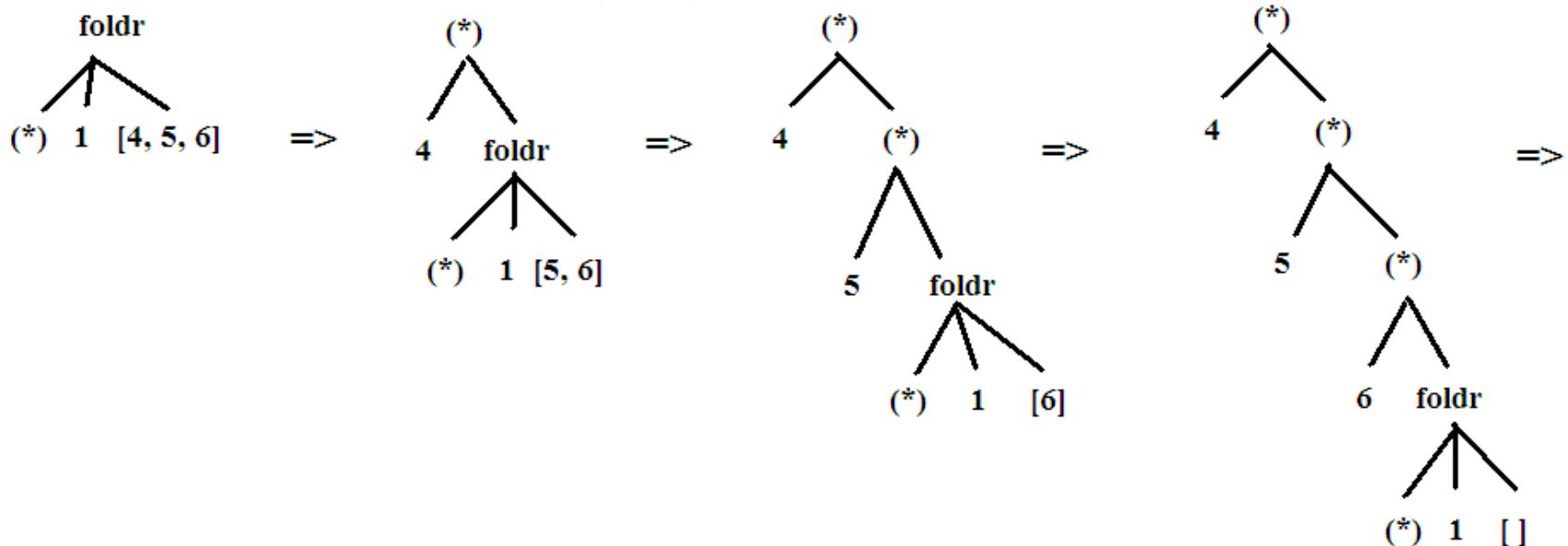
$$\begin{aligned} h [] &= e \\ h (x:xs) &= x @ h xs \end{aligned}$$

- Această funcție transformă lista $x1:(x2:(x3:(x4:[])))$ în $x1@(x2@(x3@(x4@e)))$
- Şablonul din definiţia lui h este capturat în funcţia:
 $\text{foldr} :: (\text{a} \rightarrow \text{b} \rightarrow \text{b}) \rightarrow \text{b} \rightarrow [\text{a}] \rightarrow \text{b}$
 $\text{foldr } f \ e \ [] = e$
 $\text{foldr } f \ e \ (x:xs) = f \ x \ (\text{foldr } f \ e \ xs)$
- Astfel $h = \text{foldr } (@) \ e$
- $\text{foldr } (@) \ e [x_0, x_1, \dots, x_n] = x_0 @ (x_1 @ (\dots (x_n @ e) \dots))$

λ

$\text{foldr} :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$
 $\text{foldr } f \ e \ [] = e$
 $\text{foldr } f \ e \ (x:xs) = f \ x \ (\text{foldr } f \ e \ xs)$

$\text{foldr } (*) \ 1 \ [4, 5, 6]$





Exemple

```
concat = foldr (++) []
sum = foldr (+) 0
product = foldr (*) 1
and = foldr (&&) True
length = foldr oneplus 0
    where oneplus x n = 1+n
length = foldr (\ _ n -> 1+n) 0
reverse = foldr snoc []
    where snoc x xs = xs++[x]
map f = foldr (cons.f) []
    where cons x xs = x:xs
```



Funcția foldl (fold left)

- Utilizată pentru simplificarea definițiilor recursive de forma (@ este o operație):

$$\begin{aligned} h \ e \ [] &= e \\ h \ e \ (x:xs) &= h(e @ x) xs \end{aligned}$$

- Această funcție transformă lista $x_1:(x_2:(x_3:(x_4:[])))$ în $((e@x_1)@x_2)@x_3)@x_4$
- Şablonul din definiția lui h este capturat în funcția:

$$\begin{aligned} \text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\ \text{foldl } f \ e \ [] = e \\ \text{foldl } f \ e \ (x:xs) = \text{foldl } f \ (f \ e \ x) \ xs \end{aligned}$$

- Astfel $\text{h} = \text{foldl } (@) \ e$
- $\text{foldl } (@) e [x_0, x_1, \dots, x_n] = (\dots ((e @ x_0) @ x_1) \dots x_{n-1}) @ x_n$



foldr1, foldl1

```
Prelude> :i foldr1
```

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

```
Prelude> foldr1 (*) [2,4,6]
```

```
48
```

```
Prelude> :i foldl1
```

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

```
Prelude> foldl1 (*) [2,4,6]
```

```
48
```



Exemple

```
sum = foldl (+) 0
```

```
sum =      suml 0
```

where

```
    suml e [] = e
```

```
    suml e (x:xs) = suml (e+x) xs
```

```
product = foldl (*) 1
```

```
or = foldl (||) False
```

```
and = foldl (&&) True
```

```
length = foldl (\n _ -> n+1) 0
```

```
reverse = foldl (\xs x -> x:xs) []
```

```
(xs++) = foldl (\ys y -> ys++[y]) xs
```



Funcțiile `scanl`, `scanr`

- Funcțiile **fold** “acumulează” valorile dintr-o listă returnând valoarea (una singură) obținută
- Funcția **map** aplică fiecărui element al listei o funcție, returnând toate valorile obținute
- Funcțiile **scan** “acumulează” ca și fold dar returnează lista valorilor intermediare:



Funcțiile `scanl`, `scanr`

– `scanl :: (a -> b -> a) -> a -> [b] -> [a]` acumulează de la stânga conform cu funcția din `a -> b -> a` și al doilea argument (valoarea inițială) care devine primul din lista returnată

- `scanl (+) 0 [1,2,3] = [0,1,3,6]`
- `scanl (+) 0 [] = [0]`

– `scanl1 :: (a -> a -> a) -> [a] -> [a]`, la fel ca `scanl`; valoarea inițială este primul element al listei:

- `scanl1 :: (a -> a -> a) -> [a] -> [a]`
- `scanl1 (+) [1,2,3] = [1,3,6]`
- `scanl1 (+) [] = []`



Funcțiile `scanl`, `scanr`

- `scanr :: (a -> b -> b) -> b -> [a] -> [b]` acumulează de la dreapta (conform cu funcția din $a \rightarrow b \rightarrow b$ și al doilea argument (valoarea inițială) care devine ultimul din lista returnată. Analog `scanr1`

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

```
scanr (+) 0 [1,2,3] = [6,5,3,0]
```

```
scanr (+) 0 [] = [0]
```

```
scanr1 :: (a -> a -> a) -> [a] -> [a]
```

```
scanr1 (+) [1,2,3] = [6,5,3]
```

```
scanr1 (+) [] = []
```



Studiu de caz 1: **maxlist**

- Elementul maxim dintr-o listă de elemente ce aparțin unui tip din clasa Ord:

maxlist :: (Ord a) => [a] -> a

maxlist = foldr (**max**) e

max :: Ord a => a -> a -> a

- Ce valoare se alege pentru e?
- Trebuie să fie corect:

maxlist [x] = (x `max` e) = x

- Trebuie ca e să fie cea mai mică valoare a tipului a. Există această valoare?



Studiu de caz: maxlist

- Haskell are o clasă numită **Bounded** care este constituită din tipuri cu valori mărginite:

```
Hugs.Base> :info Bounded
-- type class
class Bounded a where
    minBound :: a
    maxBound :: a

-- instances:
instance Bounded ()
instance Bounded Char
instance Bounded Int
instance Bounded Bool
instance Bounded Ordering
-----
Prelude> :i Bounded
class Bounded a where
    minBound :: a
    maxBound :: a
        -- Defined in GHC.Enum
```



Studiu de caz: maxlist

- Definim **maxlist** folosind clasa **Bounded** :

```
maxlist :: (Ord a, Bounded a) => [a] -> a
maxlist = foldr (max) minBound
```

```
Main> maxlist['a', '4', 'd']
'd'
```

```
Main> maxlist "aseewweereee"
'w'
```

```
Main> maxlist [2*x*x+5*x-1::Int| x<-[-3..4]]
51
```

```
Main> maxlist [-2*x*x+5*x-1::Int| x<-[-3..4]]
2
```



Studiu de caz: maxlist

- Soluția alternativă (fără a folosi Bounded) este de a folosi **foldr1** sau **foldl1** (doar pentru liste nevide):

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f (x::xs) = if null xs then x else f x (foldr1 f xs)
```

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x::xs) = foldl f x xs
```

```
foldr1 ( $\oplus$ ) [x0,x1,x2,x3] = x0 $\oplus$ (x1 $\oplus$ (x2  $\oplus$ x3))
foldr1 (/) [1, 2, 3] = 1.5
```

```
foldl1 ( $\oplus$ ) [x0,x1,x2,x3] = ((x0 $\oplus$ x1) $\oplus$ x2) $\oplus$ x3
foldl1 (/) [1, 2, 3] = 0.1666666666666667
```

```
maxlist1 :: (Ord a) => [a] -> a
maxlist1 = foldr1 (max)
```

```
Main> maxlist1[-x*x-x-1::Int| x<-[-5..5]]
-1
```



Cursul 5 - 6 Plan

- Studii de caz
- IO – Introducere
 - `putStr`, `putStrLn`, `getLine`, `readIO`, `readLn`,
`readFile`, `print`, `getChar`, `getLine`
 - `unlines`, `unwords`
 - Combinarea acțiunilor
 - Exemple
 - variabila `it`
- Tipuri utilizator
 - Enumerări
 - Tipuri algebrice, structuri (data)
 - Redenumiri ale tipurilor (type)
- Arbori binari
 - Definirea tipului
 - Funcții importante



Studiu de caz 2: Conversii

Proiectați o funcție **convert** care să poată fi aplicată unui întreg pozitiv **n** cu un număr de *cel mult 6 cifre*; **convert n** este lista caracterelor ce constituie pronunția lui **n** în limba română (sau în limba engleză). De exemplu:

convert 308 000

= “trei sute opt mii”

(= “three hundred and eight thousand”)

convert 313 407

= “trei sute treisprezece mii patru sute sapte”



- Listele cu denumirea numerelor:

```
unitati, sprezice, zeci :: [String]
```

```
unitati = ["unu", "doi", "trei", "patru", "cinci",  
          "sase", "sapte", "opt", "noua"]
```

```
sprezice =  
          ["zece", "unsprezice", "douasperezice", "treisprezice",  
           "patrusprezice", "cincisprezice", "sasesprezice",  
           "saptesprezice", "optsprezice", "nouasperezice"]
```

```
zeci =  
       ["douazeci", "treizeci", "patruzeci", "cincizeci",  
        "sasezeci", "saptezeci", "optzeci", "nouazeci"]
```



Cazul numerelor cu două cifre

- Pentru a converti un număr cu două cifre în stringul corespunzător, mai întâi se descompune în cele 2 cifre apoi se aleg denumirile corespunzătoare:

```
convert2 :: Int -> String
convert2 = combine2.digit2
```

```
digit2 :: Int -> (Int, Int)
digit2 n = (n `div` 10, n `mod` 10)
```

```
combine2 :: (Int, Int) -> String
combine2(0, u+1) = unitati !! u
combine2(1, u) = sprezice !! u
combine2(t+2, 0) = zeci !! t
combine2(t+2, u+1) = zeci !! t ++ " si " ++ unitati !! u
```



Cazul numerelor cu trei cifre

- Un număr de 3 cifre se descompune în partea sutelor(o cifră) și cea a zecilor(2 cifre) după care se alege sirul corespunzător folosind și **convert2**:

```
convert3 :: Int -> String  
convert3 = combine3.digit3
```

```
digit3 :: Int -> (Int, Int)  
digit3 n = (n `div` 100, n`mod` 100)
```

```
combine3 :: (Int, Int) -> String  
combine3(0, u+1) = convert2(u + 1) -- u+1 /= 0  
combine3(1, 0) ="una suta" -- unu -> una suta  
combine3(1, u+1) ="una suta " ++ convert2(u+1) -- unu -> una suta  
combine3(2, 0) ="doua sute" -- doi -> doua sute  
combine3(2, u+1) ="doua sute " ++ convert2(u+1) -- u+1 /= 0  
combine3(t+2, 0) = unitati!!(t+1) ++ " sute " -- trei... sute  
combine3(t+2, u+1) = unitati!!(t+1) ++ " sute " ++  
                           convert2(u+1)
```



Cazul numerelor cu șase cifre

- Numărul de 6 cifre se descompune în două de câte 3 cifre și se aplică corespunzător **convert3**:

```
convert6 :: Int -> String
```

```
convert6 = combine6.digit6
```

```
digit6 :: Int -> (Int, Int)
```

```
digit6 n = (n `div` 1000, n`mod` 1000)
```

```
combine6 :: (Int, Int) -> String
```

```
combine6(0, u+1) = convert3(u + 1) -- u+1 /= 0
```

```
combine6(1, 0) = "o mie "
```

```
combine6(1, u+1) = "o mie si " ++ convert3(u+1)
```

```
combine6(2, 0) ="doua mii "
```

```
combine6(2, u+1) ="doua mii si " ++ convert3(u+1)
```

```
combine6(t+2, 0) = convert3(t+2) ++ " mii"
```

```
combine6(t+2, u+1) = convert3(t+2) ++ " mii " ++
                                         convert3(u+1)
```



Cazul general

```
convert :: Int -> String
convert = convert6

Main> convert 313407
"trei sute treisprezece mii patru sute sapte"
Main> convert 308000
"trei sute opt mii"
Main> convert 101001
"una suta unu mii unu"
Main> convert 207
" doua sute sapte"
Main> convert 307
"trei sute sapte"
Main> convert 30775
"treizeci mii sapte sute saptezeci si cinci"
Main> convert 35
"treizeci si cinci"
Main> convert 1
"unu"
```

- Exercițiu: Tratați exceptiile (0, numere negative)



Introducere în IO

- Operațiile de intrare/ieșire în Haskell sunt construite în aşa fel încât să nu apară “**side-effects**”
- Sunt valori de tip **IO a** (acțiuni) unde **a** este tipul rezultatului acțiunii
- Acțiunile fără rezultat semnificativ/util sunt de tip **IO ()**



Input/Output (I/O)

- I/O sunt modelate în Haskell ca **acțiuni** sau **calcule**
- Acțiunile sunt valori de tip **IO a**
- Tipul **IO a** este un tip ce realizează acțiuni de intrare și/sau ieșire după care “returnează” o valoare de tip **a**
- Un program (în întregime) în Haskell este o valoare de tip **IO ()**
 - **()** este tipul fără nici o valoare
 - Aceasta este tipul funcției **main**



Introducere în IO

- Construirea și compilarea unui program:
 - Se definește funcția **main** de tip **IO ()**
 - Se salvează într-un fișier, de ex. **filename.hs**
 - Se compilează :
Prelude> :!ghc -o progname filename.hs
 - Se execută programul:
Prelude> :!progname
 - ctr-C îintrerupe execuția (dacă programul nu se termină)



Acțiuni I/O

- Preia un string (argument), îl afișează și nu întoarce nimic:

putStr :: **String** → **IO ()**

- Preia un string (argument), îl afișează + newline și nu întoarce nimic:

putStrLn :: **String** → **IO ()**

- Preia un string (de la tastatura) până la newline și îl returnează:

getLine :: **IO String**



```
putStr :: String -> IO ()  
putStrLn :: String -> IO ()  
getLine :: IO String  
readIO :: (Read a) => String -> IO a  
readLn :: (Read a) => IO a  
readFile :: FilePath -> IO String  
print :: Show a => a -> IO ()  
getChar :: IO Char  
getLine :: IO String
```



Exemple

- Fișier hello1.hs

```
main = putStrLn "Hello, world!"
```

- Fisier hello2.hs

```
main = do  
    x <- putStrLn "Please enter your name: "  
    name <- getLine  
    putStrLn ("Hello, " ++ name ++ ", how are  
    you?")
```



lines, unlines, words, unwords

- `lines` : It takes a string and returns every line of that string in a separate list.
- `unlines`: is the inverse function of `lines`. It takes a list of strings and joins them together using a '`\n`'.
- `words` and `unwords`: are for splitting a line of text into words or joining a list of words into a text.



```
ghci> lines "first line\nsecond line\nthird line"  
["first line","second line","third line"]
```

```
ghci> unlines ["first line", "second line", "third line"]  
"first line\nsecond line\nthird line\n"
```

```
ghci> words "hey these are the words in this sentence"  
["hey","these","are","the","words","in","this","sentence"]  
ghci> words "hey these      are    the words in this\nsentence"  
["hey","these","are","the","words","in","this","sentence"]  
ghci> unwords ["hey","there","mate"]  
"hey there mate"
```

unlines, unwords, putStrLn

```
Prelude> :t unlines
```

```
unlines :: [String] -> String
```

```
Prelude> :t unwords
```

```
unwords :: [String] -> String
```

```
Prelude> unlines ["Branza", "Lapte", "Oua"]
```

```
"Branza\nLapte\nOua\n"
```

```
Prelude> unwords ["Branza", "Lapte", "Oua"]
```

```
"Branza Lapte Oua"
```

```
Prelude> putStrLn (unlines ["Branza", "Lapte", "Oua"])
```

```
Branza
```

```
Lapte
```

```
Oua
```

```
Prelude> putStrLn (unwords ["Branza", "Lapte", "Oua"])
```

```
Branza Lapte Oua
```



```
module StringManip where
import Data.Char
uppercase, lowercase :: String -> String
uppercase = map toUpper
lowercase = map toLower
capitalise :: String -> String
capitalise x =
    let capWord [] = []
        capWord (x:xs) = toUpper x : xs
    in unwords (map capWord (words x))

*StringManip> uppercase "anabsvdv"
"ANABSVDV"
*StringManip> capitalise "alabala"
"Alabala"
```



Actiuni - 1

```
main = do
    putStrLn "Please enter your name: "
    getLine
    putStrLn ("Hello, how are you?")
```

Ok, modules loaded: Main.

*Main> main

Please enter your name:

grig

Hello, how are you?



Acțiuni - 2

```
main = do
    x <- putStrLn "Please enter your name: "
    name <- getLine
    putStrLn ("Hello, " ++ name ++ ", how are you?")
```

Ok, modules loaded: Main.

*Main> main

Please enter your name:

grig

Hello, grig, how are you?



Combinarea acțiunilor

- Este de la sine înțeleasă necesitatea combinării acțiunilor I/O pentru a obține acțiuni complexe
- Se folosesc 2 funcții de bază care characterizează clasa **Monad** (amănuite mai târziu):

```
return :: a -> IO a
```

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- **(>>=)** se numește **bind**



Combinarea acțiunilor

- **return x** convertește o valoare într-o acțiune ce returnează acea valoare
- **(>>=) :: IO a -> (a -> IO b) -> IO b** combină
 - O acțiune ce returnează o valoare de tip a
 - O funcție care ia un argument de tip a (cel returnat de acțiunea precedentă!) și returnează o acțiune ce returnează o valoare de tip b
- și obține o acțiune returnând o valoare de tip b
- **f1 >>= \x -> f2 x -- sau: f1 >>= f2**



Exemple

```
Prelude > return 9  
9  
Prelude > return "hello!"  
"hello!"  
Prelude > return [1,2,3,4,5]  
[1,2,3,4,5]  
Prelude> return "hello!" >>= \x -> putStrLn x  
hello!  
Prelude> return "hello!" >>= putStrLn  
hello!  
Prelude> do {s <- return "hello!"; putStrLn s}  
hello!
```



Exemple

```
getTwoLines :: IO String
getTwoLines = getLine >>= \a ->
               getLine >>= \b ->
               return (a ++ b)
```

```
Prelude> :l bind.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> getTwoLines
Facultatea de
    Informatica
"Facultatea de Informatica"
```



Exemple

```
Prelude> do { s1 <- getLine; s2 <- getLine; return  
    (s1++s2) }
```

Universitatea Cuza,

Facultatea de Informatica

"Universitatea Cuza, Facultatea de Informatica"

```
Prelude > do {x <- readLn; y <- readLn ; return  
    (x+y) }
```

33

99

132



Exemple

```
getTwoInts :: IO Int
getTwoInt =  readLn >>= \a ->
              readLn >>= \b ->
              return (a + b)

*Main> :r
[1 of 1] Compiling Main           ( bind.hs ,
interpreted )
Ok, modules loaded: Main .
*Main> getTwoInt
22
55
77
```



Variabila it

- Afişarea valorii unei expresii, are ca efect legarea acestei valori de variabila it:
 - Dacă după verificarea tipului expresiei acesta nu este un tip IO atunci are loc:
`let it = e; print it`
 - Dacă după verificarea tipului expresiei acesta este un tip IO atunci are loc:
`it <- e`

```
Prelude> let a = 99-88
```

```
Prelude> a
```

```
11
```

```
Prelude> it
```

```
11Prelude> 66^22
```

```
10714368571740915734427767689504050118656
```

```
Prelude> it
```

```
10714368571740915734427767689504050118656
```



it

```
Prelude Data.Time> import Data.Time.Calendar.Easter
Prelude Data.Time Data.Time.Calendar.Easter> orthodoxEaster 2015
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package old-locale-1.0.0.6 ... linking ... done.
Loading package time-1.4.2 ... linking ... done.2015-04-12
Prelude Data.Time Data.Time.Calendar.Easter> it
2015-04-12
Prelude Data.Time Data.Time.Calendar.Easter> gregorianEaster 2015
2015-04-05
Prelude Data.Time Data.Time.Calendar.Easter> it
2015-04-05
```



Definirea de tipuri de date

- Haskell are trei modalități de a introduce tipuri (tipuri utilizator):
 - Declarații **data** pentru *enumerări și structuri* (abstractive) de date
 - Declarația **type** pentru *sinonime*
 - Declarația **newtype** pentru *tipuri noi*



Enumerări

- Enumerări:
 - Enumerarea explicită a valorilor tipului
 - Tipul Bool, de exemplu, este introdus prin enumerare
data Bool = False | True
- Nu pot fi definite noi tipuri în GHCi; se definesc în cadrul unui program (script), se încarcă fișierul sau se compilează



Enumerări

```
data Triunghi = Esec|Isoscel|Echilateral|Scalen
analiza::(Int, Int, Int) -> Triunghi
analiza(x,y,z) | x+y<=z          = Esec
                | x==z            = Echilateral
                | (x==y) || (x==z) = Isoscel
                | otherwise         = Scalen
```

- Funcția **analiza** este corectă ?



Enumerări

- Exemple:

```
data Zi = Lu|Ma|Mi|Jo|Vi|Sa|Du
```

- Tipul Zi are 8 valori, cele enumerate plus **undefined**
- Cele 7 constante se numesc **constructorii** tipului Zi
- Constructorii unui tip, ca și numele tipului, trebuie să înceapă cu literă mare



Enumerări

- Elementele unui tip enumerare pot fi *comparate* dacă se declară tipul ca fiind instanță a claselor **Eq** (`==`, `/=`) și **Ord** (`<`, `<=`, `>`, `>=`)

- Exemplu:

```
data Bool = False|True
instance Eq Bool where
    (x==y) = (x&&y) || (`not` x && `not` y)
    (x/=y) = `not` (x==y)
```

- Pentru enumerările cu număr mare de valori nu este convenabilă declararea ca instanță în acest mod (pentru definirea operatorilor `==` și `/=`)



Enumerări

- Soluția: o clasă **Enum** care are ca metodă transformarea valorilor în întregi (**fromEnum**) și compararea întregilor:

```
class Enum a where
    succ :: a -> a
    pred :: a -> a
    toEnum :: Int -> a
    fromEnum :: a -> Int
    enumFrom :: a -> [a]
    enumFromThen :: a -> a -> [a]
    enumFromTo :: a -> a -> [a]
    enumFromThenTo :: a -> a -> a -> [a]
```



Enumerări

- Declarăm tipul Zi ca fiind instanță a clasei Enum:

```
instance Enum Zi where
    fromEnum Du = 0
    fromEnum Lu = 1
    fromEnum Ma = 2
    fromEnum Mi = 3
    fromEnum Jo = 4
    fromEnum Vi = 5
    fromEnum Sa = 6
```

- Declarăm Zi instanță a claselor Eq și Ord:

```
instance Eq Zi where
    (x==y)=(fromEnum x == fromEnum y)
instance Ord Zi where
    (x<y)=(fromEnum x < fromEnum y)
```



Enumerări – optimizarea definiției

- Declararea automată ca instanță a unor clase (**deriving**); sistemul generează metodele clasei pentru această instanță:

```
data Zi = Du|Lu|Ma|Mi|Jo|Vi|Sa  
deriving (Eq, Ord, Enum, Show)
```

```
zilucr::Zi -> Bool
```

```
zilucr d = (Lu <= d) && (d <= Vi)
```

```
maine:: Zi -> Zi
```

```
maine d = toEnum((fromEnum d + 1) `mod` 7)
```



Enumerări

```
Main> maine Lu
```

```
Ma
```

```
Main> zilucr Mi
```

```
True
```

```
Main> zilucr Du
```

```
False
```

```
Main> maine Du
```

```
Lu
```



Tipuri parametrize

```
data MaybeInt = NoInt | AnInt Int
```

```
data Anniversary = Birthday String Int Int Int  
                  | Wedding String String Int Int Int
```

- Tip nou : **Anniversary**
- Constructori: **Birthday**, **Wedding** (se comportă ca niște funcții)

```
ionPopa :: Anniversary  
ionPopa = Birthday "Ion Popa" 1988 7 3
```

```
popaWedding :: Anniversary  
popaWedding = Wedding "Ion Popa" "Ana Popas" 2005 3 8
```



Polimorfism (din nou)

```
data MaybeInt = NoInt | AnInt Int
```

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

```
Nothing :: Maybe a
```

```
Just 10 :: Maybe Int
```

```
Just "hi there!" :: Maybe String
```

```
Just :: a -> Maybe a
```

```
Prelude> map Just [1..5]
```

```
[Just 1,Just 2,Just 3,Just 4,Just 5]
```

```
Prelude> map Just ['a'..'e']
```

```
[Just 'a',Just 'b',Just 'c',Just 'd',Just 'e']
```



Tipuri parametrize

```
showAnniversary :: Anniversary -> String
```

```
showAnniversary (Birthday name year month day) =  
name ++ " born " ++ showDate year month day
```

```
showAnniversary (Wedding name1 name2 year month day) =  
name1 ++ " married " ++ name2 ++ " " ++ showDate year month day
```

```
showDate :: Int Int Int -> String  
showDate year month day =  
    show(year) ++ "/" ++ show(month) ++ "/" ++ show(day)
```

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

```
lookupBirthday ::  
    [Anniversary] -> String -> Maybe Anniversary
```

- funcție care are valoare Just u (u este înregistrarea găsită în listă) sau Nothing în cazul în care nu se află nici o înregistrare cu numele căutat.



data: forma generală

Declarația:

data [cx =>] T u₁ ... u_k = K₁ t₁₁...t_{1k1} | ... | K_n t_{n1}...t_{nkn}

- introduce un nou constructor de tip T
 - unul sau mai mulți constructori de dată constituenți K₁, ..., K_n
 - t_{ij} – tipuri, cx - context
- Tipul constructorilor de dată (funcții):
 $K_i :: t_{i1} \rightarrow t_{i2} \rightarrow \dots t_{iki} \rightarrow (T u_1 u_2 \dots u_k)$ (în contextul cx)

• Exemple:

```
data A = M | N
```

```
data C = F Int Int Bool
```

```
data Eq a => Set a = NilSet|ConsSet a (Set a)  
NilSet :: Set a  
ConsSet :: Eq a => a -> Set a -> Set a
```



Structuri

- Se poate optimiza o definiție de forma:

```
data Point = Pt Float Float  
pointx :: Point -> Float  
pointx (Pt x _) = x  
pointy :: Point -> Float  
pointy (Pt _ y) = y
```



Structuri

- Se definește tipul astfel (etichete pentru câmpuri):

```
data Point = Pt { pointx :: Float,  
                  pointy :: Float }
```

sau:

```
data Point = Pt { pointx, pointy :: Float }  
:t pointx  
pointx :: Point -> Float  
:t pointy  
pointy :: Point -> Float
```



Structuri

- Se pot folosi etichetele câmpurilor pentru a construi valori noi:

```
Pt {pointx = 1, pointy = 2}
```

- echivalent cu:

```
Pt 1 2
```

- Definirea funcțiilor:

```
absPoint :: Point -> Float
```

```
absPoint (Pt {pointx = x, pointy = y})  
          = sqrt (x*x + y*y)
```



Tipuri sinonime

- Sintaxa: **type Nume_nou = tip**
- Determinarea rădăcinilor unei ecuații de gradul 2:
radacini :: (Float, Float, Float) -> (Float, Float)
- Alternativă:

```
type Coef = (Float, Float, Float)
type Radacini = (Float, Float)
radacini :: Coef -> Radacini
```

```
type PozitieInPlan = (Float, Float)
type Unghi = Float
type Distanță = Float
type Pereche = (a, a)
type Automorfism = a -> a
```



Tipuri noi

- **Tipurile sinonime nu sunt tipuri noi:** metodele pentru acest tip sunt cele de la tipul pe care-l numește
- Uneori este necesar a schimba înțelesul unor metode: două unghiuri sunt egale dacă ele sunt egale modulo $2n\pi$ ($-\pi == 3\pi$).
- Soluția: tip nou și nu sinonim:

```
data Unghi = MkUnghi Float
instance Eq Unghi where
    MkUnghi x == MkUnghi y = norm x == norm y

norm :: Float -> Float
norm x
    | x < 0          = norm(x + per)
    | x >= per       = norm(x - per)
    | otherwise       = x
where per = 2*pi
```



- Studiu de caz: Arbori binari



Structura de date “arbori binari”

- O valoare a tipului **Btree** a este fie:
 - un nod frunză (Leaf) ce conține o valoare de tip a
 - un nod ramificație (Fork) și doi noi arbori, subarborele stâng al nodului ramificație respectiv cel drept
 - o frunză se numește nod exterior
 - un nod ramificație se numește nod interior
 - Btree – **constructor de tip**, Leaf, Fork – **constructori de dată**



Structura de date arbori binari

- Sintaxa pentru tipul **Btree** a este:

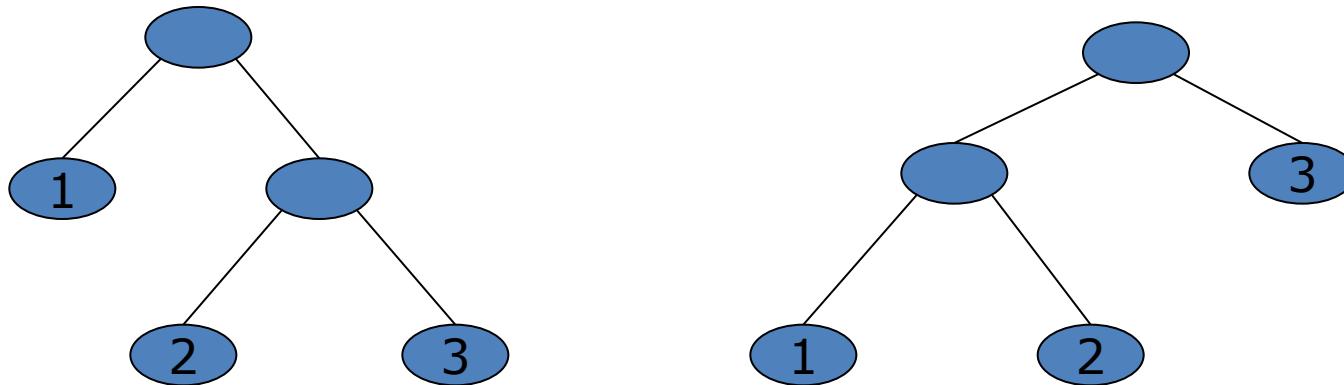
```
data Btree a = Leaf a | Fork(Btree a) (Btree a)  
deriving (Show)
```

```
Leaf :: a -> Btree a
```

```
Fork :: Btree a -> Btree a -> Btree a
```

```
Fork(Leaf 1) (Fork(Leaf 2) (Leaf 3))
```

```
Fork(Fork(Leaf 1) (Leaf 2)) (Leaf 3)
```





Structura de date arbori binari

- Variabilele ce desemnează arbori binari le vom nota xt , yt , ...
- Demonstrarea unei propoziții $P(xt)$ se face prin inducție structurală:
 - $P(\text{Leaf } x)$
 - $P(xt), P(yt)$ $P(\text{Fork } xt \ yt)$
- Un arbore finit este un arbore ce are un număr finit de frunze



Măsuri în Btree

- `size` : numărul nodurilor frunză

```
size :: Btree a -> Int
```

```
size(Leaf x) = 1
```

```
size(Fork xt yt) = size xt + size yt
```

- Legătura cu `length` de la liste:

```
size = length.flatten
```

```
flatten :: Btree a -> [a]
```

```
flatten(Leaf x) = [x]
```

```
flatten(Fork xt yt) = flatten xt ++ flatten yt
```



Exemple

```
t1, t2, t3 :: Btree Int
t1 = Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3))
t2 = Fork(Fork(Leaf 1)(Leaf 2))(Leaf 3)
t3 = Fork(Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3)))
  (Fork(Fork(Leaf 4)(Leaf 5))(Leaf 6))

Main> size t1
3
Main> size t2
3
Main> flatten t1
[1,2,3]
Main> size t3
6
Main> flatten t3
[1,2,3,4,5,6]
Main> (length.flatten) t3
6
```



Măsuri în Btree

- nodes: numărul nodurilor interne

```
nodes :: Btree a -> Int
nodes(Leaf x) = 0
nodes(Fork xt yt) = 1 + nodes xt + nodes yt
```

- height: lungimea drumului maxim de la radacină la frunze

```
height :: Btree a -> Int
height(Leaf x) = 0
height(Fork xt yt) =
    1 + (max (height xt) (height yt))
```



Exemple

```
t1, t2, t3 :: Btree Int  
t1 = Fork(Leaf 1) (Fork(Leaf 2) (Leaf 3))  
t2 = Fork(Fork(Leaf 1) (Leaf 2)) (Leaf 3)  
t3 = Fork(Fork(Leaf 1) (Fork(Leaf 2) (Leaf 3)))  
     (Fork(Fork(Leaf 4) (Leaf 5)) (Leaf 6))
```

```
Main> nodes t1
```

```
2
```

```
Main> nodes t3
```

```
5
```

```
Main> height t1
```

```
2
```

```
Main> height t3
```

```
3
```



Măsuri în Btree

- depths: funcție ce înlocuiește într-un arbore valoarea din fiecare frunză cu adâncimea (depth) frunzei în arbore
- În acest fel, înălțimea unui arbore este maximum din adâncimile frunzelor

```
depths :: Btree a -> Btree Int
```

```
depths = down 0
```

```
down :: Int -> Btree a -> Btree Int
```

```
down n (Leaf x) = Leaf n
```

```
down n (Fork xt yt) = Fork (down (n+1) xt) (down (n+1) yt)
```



Măsuri în Btree

- Înălțimea unui arbore este maximum din adâncimile frunzelor:

```
height :: Btree a -> Int
```

```
height = maxBtree.depths
```

```
depths :: Btree a -> Btree Int
```

```
maxBtree :: (Ord a) => Btree a -> a
```

```
maxBtree (Leaf x) = x
```

```
maxBtree (Fork xt yt) =
```

```
    max (maxBtree xt) (maxBtree yt)
```



Exemple:

```
t4 = Fork(Fork(Leaf '1') (Fork(Leaf '2') (Leaf '3')))  
      (Fork(Fork(Leaf '4') (Leaf '5'))) (Leaf '6'))
```

```
Main> flatten t4
```

```
"123456"
```

```
Main> (maxBtree.depths) t4
```

```
3
```

```
Main> t4
```

```
Fork (Fork (Leaf '1') (Fork (Leaf '2') (Leaf '3')))  
      (Fork (Fork (Leaf '4') (Leaf '5'))) (Leaf '6'))
```

```
Main> depths t4
```

```
Fork (Fork (Leaf 2) (Fork (Leaf 3) (Leaf 3)))  
      (Fork (Fork (Leaf 3) (Leaf 3)) (Leaf 2))
```



Arbore perfecti

- Un arbore binar se zice *perfect* dacă toate frunzele sale au aceeași adâncime

```
Main> t5
Fork (Fork (Leaf 1) (Leaf 2)) (Fork (Leaf 3) (Leaf 4))
Main> depths t5
Fork (Fork (Leaf 2) (Leaf 2)) (Fork (Leaf 2) (Leaf 2))
Main> (maxBtree.depths) t5
2
Main> flatten t5
[1,2,3,4]
```



Proprietăți

- Dacă xt este un arbore perfect atunci size xt este o putere a lui 2; există exact un arbore perfect pentru fiecare putere a lui 2 (modulo valorile frunzelor)
- $\text{height xt} < \text{size xt} \leq 2^{\text{height xt}}$
- $\lceil \log(\text{size xt}) \rceil \leq \text{height xt} < \text{size xt}$



Construcția unui arbore

- Dată o listă xs de lungime n să se construiască un arbore xt pentru care:

flatten xt = xs , height xt = log n

- Există mai mulți arbori cu această proprietate
- O soluție: se împarte xs în două și se construiește recursiv, pentru fiecare jumătate câte un arbore



Construcția unui arbore

```
mkBtree :: [a] -> Btree a
mkBtree xs
| (m == 0) = Leaf(unwrap xs)
| otherwise = Fork(mkBtree ys) (mkBtree zs)
  where m = (length xs) `div` 2
        (ys, zs) = splitAt m xs
        unwrap[x] = x
```

- De la liste:

```
splitAt n xs = (take n xs, drop n xs)
```



Construcția unui arbore: Exemple

```
Main> mkBtree [1]
Leaf 1

Main> mkBtree [1,2,3]
Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))

Main> mkBtree [1,2,3,4]
Fork (Fork (Leaf 1) (Leaf 2)) (Fork (Leaf 3) (Leaf 4))

Main> mkBtree ['a','b','c','d','e']
Fork (Fork (Leaf 'a') (Leaf 'b')) (Fork (Leaf 'c') (Fork
(Leaf 'd') (Leaf 'e')))

Main> (flatten.mkBtree) ['a','b','c','d','e']
"abcde"
```



Funcțiile **mapBtree** și **foldBtree**

- Sunt funcțiile analoge funcțiilor map și fold de la liste
- Funcția **mapBtree**:

```
mapBtree :: (a -> b) -> Btree a -> Btree b
```

```
mapBtree f (Leaf x) = Leaf f x
```

```
mapBtree f (Fork xt yt) = Fork (mapBtree f xt) (mapBtree f yt)
```

- Proprietăți:

```
mapBtree id = id
```

```
mapBtree (f . g) = mapBtree f . mapBtree g
```

```
map f . flatten = flatten . mapBtree f
```



Funcțiile **mapBtree** și **foldBtree**

- Btree are 2 constructori:

Leaf :: a → Btree a

Fork :: Btree a → Btree a → Btree a

- Funcția foldBtree trebuie să furnizeze aplicarea a 2 funcții pentru un arbore dat:

– f :: a → b (se aplică valorilor din frunze)

– g :: b → b → b (se aplică rezultatelor aplicării lui f)

- Funcția foldBtree:

foldBtree :: (a → b) → (b → b → b) → Btree a → b

foldBtree f g (Leaf x) = f x

foldBtree f g (Fork xt yt) =

g(foldBtree f g xt) (foldBtree f g yt)



Exemplu

- Funcția mapBtree:

```
Main> mapBtree (+5) t2
Fork (Fork (Leaf 6) (Leaf 7)) (Leaf 8)
Main> mapBtree (+1) t5
Fork (Fork (Leaf 2) (Leaf 3)) (Fork (Leaf 4) (Leaf 5))
```

- Funcția foldBtree:

```
Main> foldBtree (const 1) (+) t5
4
Main> foldBtree (const 1) (+) t4
6
Main> foldBtree (id) (max) t3
6
Main> foldBtree (id) (max) t4
'6'
```



Exemplu

- Funcțiile definite la început se pot exprima cu foldBtree:

```
size    = foldBtree (const 1) (+)
height = foldBtree (const 0) (⊕)
          where m ⊕ n = 1 + (m `max` n)
flatten = foldBtree wrap (++)
          where wrap x = [x]
maxBtree = foldBtree id (max)
mapBtree f = foldBtree (Leaf.f) Fork
```



Cursul 7 - 8 Plan

- Arbori binari augmentați
- Arbori binari de căutare
 - Căutare, Sortare, Inserare, Stergere
- Arbori heap
- Tipuri de date abstracte (ADT)
 - Concepte de bază
 - Modulul – mecanism pentru implementarea unui adt
 - Exemplul1: Tipul abstract Queue
 - Specificare algebrică
 - Implementare
- Data.List, Data.Char, Data.Set, Data.Map,



- Studiu de caz: Arbori binari

λ Structura de date “arbori binari”

- O valoare a tipului **Btree** a este fie:
 - un nod frunză (Leaf) ce conține o valoare de tip a
 - un nod ramificație (Fork) și doi noi arbori, subarborele stâng al nodului ramificație respectiv cel drept
 - o frunză se numește nod exterior
 - un nod ramificație se numește nod interior
 - Btree – **constructor de tip**, Leaf, Fork – **constructori de dată**



Structura de date arbori binari

- Sintaxa pentru tipul **Btree a** este:

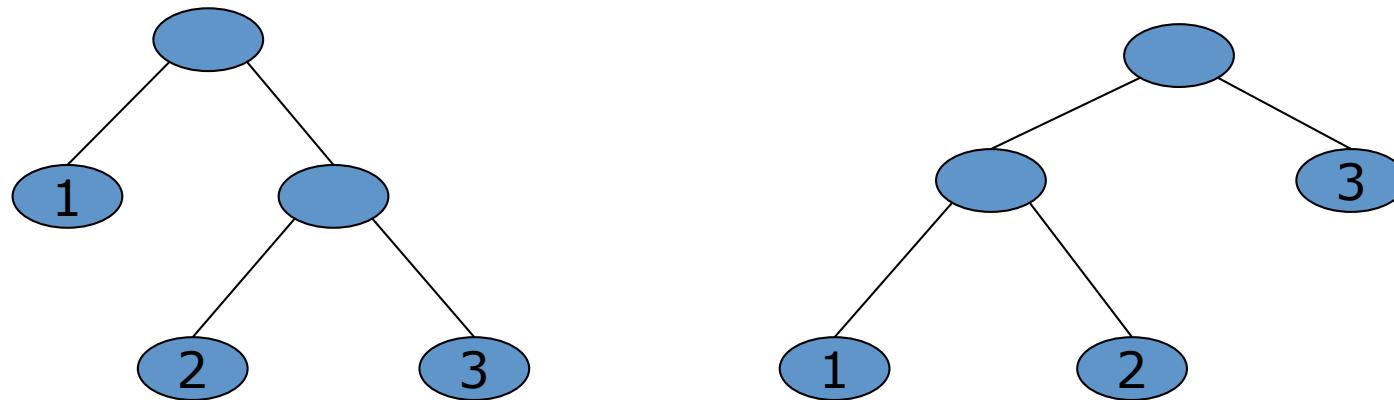
```
data Btree a = Leaf a | Fork(Btree a) (Btree a)
deriving (Show)
```

```
Leaf :: a -> Btree a
```

```
Fork :: Btree a -> Btree a -> Btree a
```

```
Fork(Leaf 1) (Fork(Leaf 2) (Leaf 3))
```

```
Fork(Fork(Leaf 1) (Leaf 2)) (Leaf 3)
```





Arborei binari augmentați

- Arborii binari introdusi au informatii doar in frunze
- Este util in aplicatii sa adaugam informatii in nodurile interioare:

```
data Atree a = Leaf a | Fork Int (Atree a)  
                  (Atree a)
```

- O idee este ca in arborele **Fork n xt yt** n sa fie **size xt + size yt**
- Acest lucru este asigurat prin construirea de noduri fork utilizand o functie adevarata



Arbore binari augmentați

- Funcția de construire a nodurilor ramificație

```
fork :: Atree a -> Atree a -> Atree a
```

```
fork xt yt = Fork n xt yt
```

```
where n = lsize xt + lsize yt
```

```
lsize :: Atree a -> Int
```

```
lsize(Leaf x) = 1
```

```
lsize(Fork n xt yt) = n
```



Arbore binari augmentați

- Funcția mkAtree:

```
mkAtree :: [a] -> Atree a
mkAtree xs
  | (m == 0)  = Leaf(unwrap xs)
  | otherwise = fork(mkAtree ys) (mkAtree zs)
    where m = (length xs) `div` 2
          (ys, zs) = splitAt m xs
          unwrap [x] = x
```



Arbore binari augmentați

- Funcția mkAtree:

```
Main> mkAtree [1,2,3,4]
```

```
Fork 4 (Fork 2 (Leaf 1) (Leaf 2)) (Fork 2 (Leaf 3)
(Leaf 4))
```

```
Main> mkAtree ['a', 'b', 'c', 'd', 'e']
```

```
Fork 5 (Fork 2 (Leaf 'a') (Leaf 'b'))
(Fork 3 (Leaf 'c')) (Fork 2 (Leaf 'd')) (Leaf 'e'))))
```



Arbore binari de căutare

- Structura de date arbore binar de căutare trebuie să fie legată de un tip din Ord:

```
data (Ord a) => Stree a = Null | Fork(Stree a) a (Stree a)
```

- Un arbore Stree nu mai are noduri frunză
- Constructorul Null denotă arborele vid
- Un arbore nevid are un subarbore stâng, o etichetă de tip a și un subarbore drept
- Arborii Stree se mai numesc arbori etichetați



Arbore binari de căutare

- Crearea unui arbore de căutare de la o secvență dată:

```
mkStree :: (Ord a) => [a] -> Stree a
mkStree [] = Null
mkStree (x:xs) = Fork(mkStree ys) x (mkStree zs)
    where (ys, zs) = partition (<= x) xs

partition :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs = (filter p xs, filter (not.p) xs)
```

- Nu se construiește arborele de adâncime minimă



Arbore binari de căutare

- Crearea unui arbore de căutare de la o secvență dată:

```
Main> mkStree [1,2,3,4,5,6,7]
Fork Null 1 (Fork Null 2 (Fork Null 3 (Fork Null 4 (Fork Null 5 (Fork
Null 6 (Fork Null 7 Null))))))

Main> mkStree [3,5,2,6,7]
Fork (Fork Null 2 Null) 3 (Fork Null 5 (Fork Null 6 (Fork Null 7
Null)))

Main> mkStree ['a','d','e','m','a']
Fork (Fork Null 'a' Null) 'a' (Fork Null 'd' (Fork Null 'e'
(Fork Null 'm' Null)))
```



Arbore binari de căutare

- Funcția de căutare **member** :

```
member :: (Ord a) => a -> Stree a -> Bool
member x Null = False
member x (Fork xt yt)    | (x < y)      = member x xt
                           | (x == y)     = True
                           | (x > y )     = member x yt
```

```
Main> member 2 (mkStree [5,3,6,1,2,4])
```

```
True
```

```
Main> member 5 (mkStree [5,3,6,1,2,4])
```

```
True
```

```
Main> member 7 (mkStree [5,3,6,1,2,4])
```

```
False
```



Arbore binari de căutare

- Funcția de sortare :

```
sort :: (Ord a) => [a] -> [a]
```

```
sort = flatten.mkStree
```

```
Main> flatten (mkStree [5,3,6,1,2,4])
```

```
[1,2,3,4,5,6]
```

```
Main> sort [5,3,6,1,2,4]
```

```
[1,2,3,4,5,6]
```

```
Main> sort [5,3,6,1,2,4,2,6]
```

```
[1,2,2,3,4,5,6,6]
```



Arbore binari de căutare-inserare

- Adăugarea unui nod de etichetă dată:
 - Dacă eticheta există atunci nu se adaugă nici un nod
 - Dacă eticheta nu există se adaugă ca și nod fără descendenți, la locul lui

```
insert :: (Ord a) => a -> Stree a -> Stree a
insert x Null = Fork Null x Null
insert x (Fork xt yt)
| (x < y) = Fork (insert x xt) y yt
| (x == y)= Fork xt y yt
| (x > y) = Fork xt y (insert x yt)
```



Arbore binari de căutare-inserare

- Exemple:

```
Main> mkStree [2,5,3,1]
```

```
Fork (Fork Null 1 Null) 2 (Fork (Fork Null 3 Null) 5 Null)
```

```
Main> insert 6 (mkStree [2,5,3,1])
```

```
Fork (Fork Null 1 Null) 2 (Fork (Fork Null 3 Null) 5 (Fork Null 6 Null))
```

```
Main> mkStree "info"
```

```
Fork (Fork Null 'f' Null) 'i' (Fork Null 'n' (Fork Null 'o' Null))
```

```
Main> insert 'a' (insert 'b' (mkStree "info"))
```

```
Fork (Fork (Fork (Fork Null 'a' Null) 'b' Null) 'f' Null) 'i' (Fork Null  
'n' (Fork Null 'o' Null))
```



Arbore binari de căutare-ștergere

- Ștergerea unui nod de etichetă dată presupune ca cei 2 subarbore rămași prin eliminarea rădăcinii să fie combinați astfel ca noul arbore să aibă proprietățile cerute
- O funcție **join** care combină în acest mod 2 arbori ar trebui să îndeplinească condiția

```
flatten(join xt yt) = flatten xt ++ flatten yt
```

- O soluție (nu cea mai bună) este să se înlocuiască cel mai din dreapta subarbore Null din xt cu yt:

```
join :: (Ord a) => Stree a -> Stree a -> Stree a  
join Null yt = yt  
join (Fork ut x vt) yt = Fork ut x (join vt yt)
```



Arbore binari de căutare-ștergere

- Exemplu:

```
Main> t1
```

```
Fork (Fork Null 1 Null) 4 (Fork Null 6 Null)
```

```
Main> t2
```

```
Fork (Fork Null 8 Null) 9 (Fork Null 11 Null)
```

```
Main> flatten(join t1 t2)
```

```
[1,4,6,8,9,11]
```

```
Main> flatten t1 ++ flatten t2
```

```
[1,4,6,8,9,11]
```

```
Main> flatten (Fork t1 7 t2)
```

```
[1,4,6,7,8,9,11]
```



Arbore binari de căutare-ștergere

- Funcția delete se poate defini astfel:

```
delete :: (Ord a) => a -> Stree a -> Stree a
delete x Null = Null
delete x (Fork xt yt)
  | (x < y) = Fork (delete x xt) y yt
  | (x == y)= join xt yt
  | (x > y) = Fork xt y (delete x yt)
```



Arbore binari de căutare-ștergere

- Exemple:

```
Main> flatten(delete 7 (Fork t1 7 t2))
[1,4,6,8,9,11]
Main> flatten(delete 4 (Fork t1 7 t2))
[1,6,7,8,9,11]
Main> flatten(delete 3 (Fork t1 7 t2))
[1,4,6,7,8,9,11]
Main> flatten(delete 1 (Fork t1 7 t2))
[4,6,7,8,9,11]
Main> flatten(delete 11 (Fork t1 7 t2))
[1,4,6,7,8,9]
Main> delete 1 (Fork t1 7 t2)
Fork (Fork Null 4 (Fork Null 6 Null)) 7 (Fork (Fork Null 8 Null) 9
    (Fork Null 11 Null))
Main> delete 3 (Fork t1 7 t2)
Fork (Fork (Fork Null 1 Null) 4 (Fork Null 6 Null)) 7 (Fork (Fork
    Null 8 Null) 9 (Fork Null 11 Null))
```



Arbore binari de căutare-ștergere

- Altă soluție pentru join: cea mai mică etichetă a arborelui yt să devină rădacină pentru noul arbore
 - Asta implementează faptul că:
$$xs \text{ ++ } ys = xs \text{ ++ } [\text{head } ys] \text{ ++ } \text{tail } ys$$
 - Folosim funcțiile **headTree** și **tailTree** cu proprietățile:

```
headTree :: (Ord a) => Stree a -> a
headTree = head.flatten
```

```
tailTree :: (Ord a) => Stree a -> Stree a
flatten.tailTree = tail.flatten
```

```
join :: (Ord a) => Stree a -> Stree a -> Stree a
join xt yt = if yt == Null then xt else
              Fork xt (headTree yt) (tailTree yt)
```



Arbore binari de căutare-ștergere

- Implementarea funcțiilor headTree și tailTree:

```
splitTree :: Ord a => Stree a -> (a, Stree a)
```

```
splitTree (Fork xt y yt) =
```

```
    if xt== Null then (y, yt) else (x, Fork wt y yt)
```

```
    where (x, wt) = splitTree xt
```

```
headTree :: (Ord a) => Stree a -> a
```

```
headTree(Fork xt y yt) = fst (splitTree (Fork xt y yt))
```

```
tailTree :: (Ord a) => Stree a -> Stree a
```

```
tailTree (Fork xt y yt) = snd (splitTree (Fork xt y yt))
```



Arbore binari de căutare-ștergere

- Exemple:

```
Main> delete 1 (Fork t1 7 t2)
Fork (Fork Null 4 (Fork Null 6 Null)) 7 (Fork (Fork
    Null 8 Null) 9 (Fork Null 11
    Null))

Main> flatten(delete 11 (Fork t1 7 t2))
[1,4,6,7,8,9]

Main> flatten(delete 11 (Fork t1 7 t2))
[4,6,7,8,9,11]

Main> flatten(delete 3 (Fork t1 7 t2))
[1,4,6,7,8,9,11]

Main> flatten(delete 4 (Fork t1 7 t2))
[1,6,7,8,9,11]
```



Arbore binari de căutare-ștergere

- Exemple:

```
Main> flatten(delete 7 (Fork t1 7 t2))
```

```
[1,4,6,8,9,11]
```

```
Main> delete 7 (Fork t1 7 t2)
```

```
Fork (Fork (Fork Null 1 Null) 4 (Fork Null 6 Null)) 8  
      (Fork Null 9 (Fork Null 11 Null))
```

```
Main> delete 4 (Fork t1 7 t2)
```

```
Fork (Fork (Fork Null 1 Null) 6 Null) 7 (Fork (Fork  
      Null 8 Null) 9 (Fork Null 11 Null))
```

```
Main> delete 9 (Fork t1 7 t2)
```

```
Fork (Fork (Fork Null 1 Null) 4 (Fork Null 6 Null)) 7  
      (Fork (Fork Null 8 Null) 11 Null)
```

```
Main> delete 6 (Fork t1 7 t2)
```

```
Fork (Fork (Fork Null 1 Null) 4 Null) 7 (Fork (Fork  
      Null 8 Null) 9 (Fork Null 11 Null))
```



Arbore binari heap

- minHeap – arbore binar în care cheia din fiecare nod este mai mică decât cheile din nodurile fiilor. Analog maxHeap
- Vom ilustra minHeap
- Structura de date:

```
data (Ord a) => Htree a = Null | Fork a (Htree a) (Htree a)
    deriving Show
```

- Htree este virtual echivalent cu Stree: etichetele la constructorul Fork sunt plasate diferit
- Pentru a pune în evidență proprietatea heap se definește corespunzător funcția **flatten**



Arbore binari heap

- Funcția flatten:

```
flatten :: (Ord a) => Htree a -> [a]
flatten Null = []
flatten (Fork x xt yt) =
    x : merge (flatten xt) (flatten yt)
```

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge (x:xs) [] = x:xs
merge (x:xs) (y:ys) = if x <= y then x:merge
    xs (y:ys) else y:merge (x:xs) ys
```



Arbore binari heap

- Exemplu:

```
Main> flatten( Fork 1 (Fork 2 Null Null) ( Fork 3 Null Null))  
[1,2,3]
```

```
Main> flatten( Fork 1 (Fork 3 Null Null) ( Fork 2 Null Null))  
[1,2,3]
```

```
Main> t1
```

```
Fork 1 (Fork 3 (Fork 4 Null Null) (Fork 5 Null Null)) (Fork 2  
Null Null)
```

```
Main> flatten t1
```

```
[1,2,3,4,5]
```

```
Main> t2
```

```
Fork 15 (Fork 18 (Fork 29 Null Null) (Fork 25 Null Null))  
(Fork 29 Null Null)
```

```
Main> flatten t2
```

```
[15,18,25,29,29]
```



heap vs. căutare

- Arborii binari de căutare utili pentru :
 - Căutare eficientă
 - Inserare
 - Ștergere
- Arborii heap:
 - Aflarea min (respectiv max) în timp constant
 - Operație “merge” eficient (combinarea a doi arbori de cautare este ineficientă)
 - Heapsort



Construirea unui heap

- Se poate defini o nouă versiune a funcției mkBtree cu îndeplinirea condiției de heap. Nu se obține însă arbore complet
- Este posibilă construirea unui arbore binar heap dintr-o listă, în timp liniar
 - Se construiește mai întâi un arbore binar de adâncime minimă (mkHtree)
 - Se rearanjează etichetele astfel încât să fie îndeplinită condiția heap (heapify)



Construirea unui heap

```
mkHtree :: (Ord a) => [a] -> Htree a
mkHtree [] = Null
mkHtree (x:xs)
| (m == 0) = Fork x Null Null
| otherwise = Fork x (mkHtree ys) (mkHtree zs)
  where m = (length xs)
        (ys, zs) = splitAt (m `div` 2) xs
```

```
mkHeap :: (Ord a) => [a] -> Htree a
mkHeap = heapify.mkHtree
```

```
heapify :: (Ord a) => Htree a -> Htree a
heapify Null = Null
heapify (Fork x xt yt) = sift x (heapify xt) (heapify yt)
```

- Funcția **sift** are 3 argumente (x xt yt); reconstruiește arborele prin deplasarea lui x “în jos” până se obține proprietatea de heap



Construirea unui heap – funcția sift

```
sift :: (Ord a) => a -> Htree a -> Htree a -> Htree a
```

```
sift x Null Null = Fork x Null Null
```

```
sift x (Fork y a b) Null =
  if x <= y    then Fork x (Fork y a b) Null
  else Fork y (sift x a b) Null
```

```
sift x Null (Fork z c d) =
  if x <= z    then Fork x Null (Fork z c d)
  else Fork z Null (sift x c d)
```

```
sift x (Fork y a b) (Fork z c d)
| x <= (y `min` z) = Fork x (Fork y a b) (Fork z c d)
| y <= (x `min` z) = Fork y (sift x a b) (Fork z c d)
| z <= (x `min` y) = Fork z (Fork y a b) (sift x c d)
```



Construirea unui heap – Exemplu

```
Main> mkHtree["unu","doi","trei","patru"]
Fork "unu" (Fork "doi" Null Null) (Fork "trei" Null (Fork "patru" Null Null))
```

```
Main> mkHeap["unu","doi","trei","patru"]
Fork "doi" (Fork "unu" Null Null) (Fork "patru" Null (Fork "trei" Null Null))
```

```
Main> mkHtree[5,7,2,4,1,3,6]
Fork 5 (Fork 7 (Fork 2 Null Null) (Fork 4 Null Null)) (Fork 1 (Fork 3 Null
Null (Fork 6 Null Null)))
Main> flatten( mkHtree[5,7,2,4,1,3,6])
[5,1,3,6,7,2,4]
```

```
Main> mkHeap[5,7,2,4,1,3,6]
Fork 1 (Fork 2 (Fork 7 Null Null) (Fork 4 Null Null)) (Fork 3 (Fork 5 Null
Null) (Fork 6 Null Null))
```

```
Main> flatten(mkHeap[5,7,2,4,1,3,6])
[1,2,3,4,5,6,7]
```



Heapsort

```
heapsort :: (Ord a) => [a] -> [a]
heapsort = flatten.mkHeap
```

```
Main> heapsort[3,1,3,5,2,1]
```

```
[1,1,2,3,3,5]
```

```
Main> heapsort ["unu", "doi","trei", "patru", "cinci",
"sase","sapte", "opt", "noua"]
```

```
["cinci", "doi", "noua", "opt", "patru", "sapte", "sase",
"trei", "unu"]
```

```
Main> heapsort [[1,2,4],[1,2,3],[1,2,1],[1,2,2]]
```

```
[[1,2,1],[1,2,2],[1,2,3],[1,2,4]]
```

```
Main> heapsort [[1,2,5,4],[2,3],[1,1,2,1],[1,2,2],[1],[3]]
[[1],[1,1,2,1],[1,2,2],[1,2,5,4],[2,3],[3]]
```



Tipuri de date abstracte

- O declarație **data** introduce un nou tip de dată prin descrierea modului de construire a elementelor sale
- Un tip în care sunt descrise valorile fără a descrie operațiile se numește **tip concret**
- Un **tip abstract de date** este definit prin specificarea operațiilor sale fără a descrie modul de reprezentare a valorilor
- Exemplu: Float este adt în Haskell pentru că nu este specificat modul de reprezentare al elementelor ci doar operațiile ce se aplică
- În general reprezentarea adt se poate schimba fără a afecta validitatea scripturilor ce utilizează acest adt



Exemplu - Coada

- Tipul abstract **Queue a** al cozilor peste un tip a
- O coadă este o listă specială: restricții privind operațiile
- Programatorul dorește o implementare eficientă a acestui adt dar nu o realizează (încă!) ci se preocupă de descrierea operațiilor
 - Operațiile primitive – numele și descrierea lor



Coadă

- Operațiile primitive – numele, tipul:
 - **empty** :: Queue a
 - **join** :: a -> Queue a -> Queue a
 - **front** :: Queue a -> a
 - **back** :: Queue a -> Queue a
 - **isEmpty** :: Queue a -> Bool
- Semnificația
- Lista operațiilor împreună cu tipul acestora reprezintă **signatura tipului de dată abstract**



Coadă

- **Specificare algebrică (axiomatică):** o listă de axiome ce trebuie să fie satisfăcute de operații
- Pentru **Queue a:**

`isEmpty empty = True`

`isEmpty (join x xq) = False`

`front (join x empty) = x`

`front (join x(join y xq)) = front (join y xq)`

`back(join x empty) = empty`

`back(join x (join y xq)) = join x(back (join y xq))`



Coadă - Specificare algebrică

- Aceste ecuații seamănă cu definițiile formale ale funcțiilor, bazate pe un tip de dată ce are constructorii **empty** și **join**
- Nu se specifică nicăieri constrângerea de a implementa coada cu acești constructori; ecuațiile trebuie exprimate ca o exprimare a relațiilor între funcțiile adt
- Din ecuațiile de mai sus se deduce că orice coadă poate fi exprimată printr-un număr finit de aplicații ale operațiilor de **join**, **front** și **back** aplicate cozii **empty**



ADT - Implementare

- Implementarea adt înseamnă:
 - furnizarea unei reprezentări pentru valorile sale
 - definirea operațiilor în termenii acestei reprezentări
 - dovedirea faptului că operațiile implementate satisfac specificațiile algebrice
- Cel ce implementează adt este liber în a alege dintre posibilele reprezentări, în funcție de :
 - eficiență
 - simplitate
 - ...gusturi



Coadă – Implementare(1)

- Reprezentarea cu **liste finite**
- Operațiile (le numim cu sufixul c pentru a le diferenția de cele abstracte):

```
emptyc :: [a]
emptyc = []
joinc :: a -> [a] -> [a]
joinc x xs = xs ++ [x]
frontc :: [a] -> a
frontc(x:xs) = x
backc :: [a] -> [a]
backc(x:xs) = xs
isEmptyc :: [a] -> Bool
isEmptyc xs = null xs
```

- Toate operațiile necesită timp constant, exceptie joinc care se face în $\Theta(n)$ pași



Coadă – Implementare(1)

- Pentru a dovedi că sunt îndeplinite axiomele se observă că există o **corespondență biunivocă** între **liste finite și cozi**:

```
abstr :: [a] -> Queue a
```

```
abstr = (foldr join empty) . reverse
```

$$\begin{aligned} \text{abstr } [] &= (\text{foldr join empty}) . \text{reverse } [] \\ &= \text{foldr join empty } [] = \text{empty} \end{aligned}$$

$$\begin{aligned} \text{abstr } [x] &= (\text{foldr join empty}) . \text{reverse } [x] \\ &= (\text{foldr join empty}) [x] \\ &= \text{join } x \text{ empty} \end{aligned}$$

$$\begin{aligned} \text{abstr } [x, y] &= \text{foldr join empty} . \text{reverse } [x, y] \\ &= \text{foldr join empty } [y, x] \\ &= \text{join } y \text{ (join } x \text{ empty)} \end{aligned}$$



```
reprn :: Queue a -> [a]
reprn empty = []
reprn (join x xq) = reprn xq ++ [x]

reprn (join x empty)          = reprn empty ++ [x]
                                = [] ++ [x] = [x]
reprn (join x (join y empty)) = reprn (join y empty)
                                ++ [x] = [y] ++ [x] = [y, x]
```

```
reprn.abstr = id[a]
abstr.reprn = idQueue a
```

```
reprn.abstr [x,y]    = reprn(abstr [x,y])
                      = reprn(join y (join x empty))
                      = [x, y]
abstr.reprn (join x (join y empty)) =
abstr [y,x] = join x (join y empty)
```



Coadă – Implementare(1)

- Să dovedim că au loc ecuațiile pentru `front`:

`front(join x empty) = x`

`front(join x empty) = front(join x []) = front [x] = x`

`front (join x(join y xq)) = front (join y xq)`

`front (join x(join y xq)) = front (join x(xq ++ [y])) =
= front (xq ++ [y] ++ [x]) = front (xq + [y])`

`front (join y xq) = front (xq ++ [y])`



Coadă – Implementare(2)

- O coadă xq se reprezintă ca **o pereche de liste (xs, ys)** astfel ca elementele lui xq sunt elementele listei xs ++ reverse ys, cu o condiție în plus: dacă în perechea (xs, ys) ce reprezintă o coadă, xs este lista vidă atunci și ys este lista vidă
- Nu orice pereche de liste reprezintă o coadă; de exemplu perechea ([], [1])
- Două perechi distințe de liste pot reprezenta aceeași coadă: ([1,2], []) și ([1], [2]) reprezintă coada join 2(join 1 empty)



Coada – Implementare(2)

- Pentru implementare definim funcția **abstr**:

```
abstr :: ([a], [a]) -> Queue a  
abstr (xs, ys) = (foldr join empty.reverse) (xs ++ reverse ys)
```

- Funcția abstr nu este injectivă: de exemplu se arată ușor că $\text{abstr}([1,2], []) = \text{abstr}([1], [2])$
- Formalizarea faptului că nu toate reprezentările sunt valide:

```
valid :: ([a], [a]) -> Bool  
valid(xs, ys) = not(null xs) `or` null ys
```

- Funcția **valid** este un **invariant** al tipului de dată
- Perechea **(abstr, valid)** formalizează reprezentarea cozilor prin perechi de liste



Coada – Implementare(2)

- Implementare operațiilor:

```
emptyc = ([] , [])
isEmptyc(xs, ys) = null xs
joinc x (ys,zs) = mkValid(ys, x:zs)
frontc(x:xs, ys) = x
backc(x:xs, ys) = mkValid(xs, ys)

mkValid :: ([a], [a]) -> ([a], [a])
mkValid (xs, ys) = if null xs then (reverse ys, [])
                    else (xs, ys)
```

- Funcția `mkValid` menține invariantul tipului de dată
- Toate operațiile necesită timp constant exceptând `back` în cazul în care `xs` este `[x]`



Coadă – Implementare(2)

- Verificarea specificării:
 - Faptul că o coadă poate avea mai multe reprezentări, verificarea axiomelor “mot-a-mot” duce la eșec. Axioma:

back(join x (join y xq)) = join x back (join y xq)

implică:

**backc(joinc x (joinc y (joinc z emptyc))) =
backc(joinc x (backcc (joinc y (joinc z (joinc u
emptyc)))))**

adică:

([y, x], []) = ([y], [x])

ceea ce nu este adevărat! Dar cele 2 perechi reprezintă aceeași coadă!



Coada – Implementare(2)

- Verificarea specificării: axioma să conducă la faptul că cele **2 perechi rezultate**, chiar dacă sunt diferite, **reprezintă aceeași coadă**:

```
abstr.backc.joinc x.joinc y =  
= abstr.joinc x.backc.joinc y
```



Coada – Implementare(2)

- În general, pentru orice axiomă de forma $f = g$ unde f și g returnează cozi, trebuie să aibă loc
 $\text{abstr.fc} = \text{abstr.gc}$
unde fc și gc sunt rezultatele obținute prin înlocuirea operațiilor abstracte cu implementările lor.
Dacă f și g returnează altceva decât cozi nu se folosește abstr
- Axiomele trebuie verificate doar pentru reprezentări valide:

$\text{abstr.fc} = \text{abstr.gc}$ (modulo valid)



Coada – Implementare(2)

- Verificarea specificării, altă abordare: este suficient să aibă loc următoarele ecuații, modulo valid:

```
abstr emptyc = empty
abstr.joinc x = join x.abstr
abstr.frontc = front.abstr
abstr.backc = back.abstr
isEmptyc = isEmpty.abstr
```

- Odată verificate acestea se dovedește că au loc axiomele. De exemplu, pentru ultima axiomă:

abstr.backc.joinc x.joinc y = back.join x.join y.abstr

abstr.joinc x.backc.joinc y = join x.back.join y.abstr

iar

back.join x = join x.back

este adevărată (axioma 2 pentru back)



Module

- **Modulul** – mecanism pentru definirea unui adt
- Sintaxa:

```
module Nume_modul(Lista_export) where
    Implementare
```

- *Nume_modul* începe cu literă mare
- *Listă_export* conține:
 - Numele tipului abstract de date – același cu numele modulului
 - Numele operațiilor
- Nici un alt nume sau valoare declarat în modul și care nu apare în lista export nu poate fi utilizat în altă parte
- Asta înseamnă că implementarea descrisă în modul este ascunsă în orice script ce folosește modulul



Exemplu: modulul Queue

```
module Queue(Queue, empty, isEmpty, join, front, back) where
newtype Queue a = MkQ([a], [a])
--deriving (Show)

empty :: Queue a
empty = MkQ([], [])

isEmpty :: Queue a -> Bool
isEmpty(MkQ(xs, ys)) = null xs

join :: a -> Queue a -> Queue a
join x (MkQ(ys,zs)) = mkValid(ys, x:zs)

front :: Queue a -> a
front(MkQ(x:xs, ys)) = x

back :: Queue a -> Queue a
back(MkQ(x:xs, ys)) = mkValid(xs, ys)

mkValid :: ([a], [a]) -> Queue a
mkValid (xs, ys) = if null xs then MkQ(reverse ys, [])
                  else MkQ(xs, ys)
```



Module - utilizare

- Utilizarea unui modul într-un script se face folosind o declaratie import în acel script:

```
import Nume_modul
```

- Exemplu: scriptul coada.hs

```
import Queue
toQ :: [a] -> Queue a
toQ = foldr join empty.reverse
fromQ :: Queue a -> [a]
fromQ q = if isEmpty q then [] else front q:fromQ(back q)

c1 :: Queue Int
c2 :: Queue [Char]
c1 = join 1(join 2(join 3(join 4(join 5 empty))))
c2 = join "ion" (join "vasile"(join "ana" empty))
```



Exemple

```
Main> join 1(join 2(join 3 empty))
MkQ ([3],[1,2])
Main> c1
MkQ ([5],[1,2,3,4])
Main> c2
MkQ (["ana"],["ion","vasile"])
Main> toQ [1,2,3,4,5]
MkQ ([1],[5,4,3,2])
Main> toQ ['a','b','c']
MkQ ("a","cb")
Main> join 8 c1
MkQ ([5],[8,1,2,3,4])
Main> join "horia" c2
MkQ (["ana"],["horia","ion","vasile"])
Main> front c2
"ana"
Main> front (back c2)
"vasile"
```



Exemple

```
Main> mkValid ([1,2,3], [4,5])
ERROR - Undefined variable "mkValid"
-- daca adaug mkValid in lista_export
Main> :r
Main> mkValid ([1,2,3], [4,5])
MkQ ([1,2,3],[4,5])
Main> mkValid ([], [2,4,5])
MkQ ([5,4,2],[])
Main> c1
MkQ ([5],[1,2,3,4])
Main> front c1
5
Main> let cc = back c1
Main> cc
MkQ ([4,3,2,1],[])
Main> isEmpty cc
False
```



Modules, Loading modules

- The syntax for importing modules in a Haskell script is **import <module name>**.
- One script can, of course, import several modules. Just put each import statement into a separate line.
- When you do import Data.List, all the functions that Data.List exports become available in the global namespace



Modules, Loading modules

```
Prelude> nub [2,3,2,3,4,3,2,4,4,2,4]
<interactive>:2:1: Not in scope: `nub'
Prelude> :m + Data.List
Prelude Data.List>nub [2,3,2,4,3,2,4,2,4]
[2,3,4]
```

- You can also put the functions of modules into the global namespace when using GHCI. If you're in GHCI and you want to be able to call the functions exported by Data.List, do this:

```
ghci> :m + Data.List
```

- If we want to load up the names from several modules:

```
ghci> :m + Data.List Data.Map Data.Set
```



Modules, Loading modules

- If you just need a couple of functions from a module, you can selectively import just those functions:

```
import Data.List (nub, sort)
```

- That's often useful when several modules export functions with the same name and you want to get rid of the offending ones:

```
import Data.List hiding (nub)
```



Modules, Loading modules

- Another way of dealing with name clashes is to do qualified imports:

```
import qualified Data.Map
```

- This makes it so that if we want to reference Data.Map's filter function, we have to do Data.Map.filter

```
import qualified Data.Map as M
```

- Now, to reference Data.Map's filter function, we just use M.filter



Data.List

- the Prelude module exports some functions from Data.List (map, filter, etc)
- **intersperse** takes an element and a list and then puts that element in between each pair of elements in the list:

```
Prelude Data.List> intersperse '.' "MONKEY"
```

```
"M.O.N.K.E.Y"
```

```
Prelude Data.List> intersperse 0 [1,2,3,4,5,6]
```

```
[1,0,2,0,3,0,4,0,5,0,6]
```



- **intercalate** takes a list of lists and a list. It then inserts that list in between all those lists and then flattens the result:

```
Prelude Data.List> intercalate " " ["hey","there","guys"]  
"hey there guys"
```

```
Prelude Data.List> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]  
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

- **transpose** transposes a list of lists. If you look at a list of lists as a 2D matrix, the columns become the rows and vice versa:

```
Prelude Data.List> transpose [[1,2,3],[4,5,6],[7,8,9]]  
[[1,4,7],[2,5,8],[3,6,9]]
```

```
Prelude Data.List> transpose ["hey","there","guys"]  
["htg","ehu","yey","rs","e"]
```



- **iterate** takes a function and a starting value. It applies the function to the starting value, then it applies that function to the result, then it applies the function to that result again, etc. It returns all the results in the form of an infinite list.

```
Prelude Data.List> take 10 (iterate (*2) 5)
```

```
[5,10,20,40,80,160,320,640,1280,2560]
```

```
Prelude Data.List> take 10 $ iterate (*2) 5
```

```
[5,10,20,40,80,160,320,640,1280,2560]
```



takeWhile, dropWhile

```
Prelude Data.List> takeWhile (>3)
[6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
Prelude Data.List> dropWhile (/=' ')
  "This is a sentence"
  " is a sentence"
Prelude Data.List> sum $ takeWhile (<10000) $ map
  (^3) [1..]
53361
```

```
Prelude Data.List> let stock = [(994.4,2008,9,1),
  (995.2,2008,9,2),(999.2,2008,9,3),
  (1001.4,2008,9,4),(998.3,2008,9,5)]
Prelude Data.List> head (dropWhile (\(val,y,m,d) ->
  val < 1000) stock)
(1001.4,2008,9,4)
```



break, span

```
Prelude Data.List> break (==4)
[1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

```
Prelude Data.List> span (==4)
[1,2,3,4,5,6,7]
([], [1,2,3,4,5,6,7])
```

```
Prelude Data.List> span (/=4)
[1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

- **break p** equivalent of **span (not . p)**.



sort, group, find

```
Prelude Data.List> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
Prelude Data.List> sort "This will be sorted soon"
" Tbdeehiillnooorssstw"
Prelude Data.List> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
Prelude Data.List> group.sort $
  [1,1,2,7,5,7,2,2,1,1,2,2,7,6,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2,2,2,2,2,2,2],[3,3],[5,5],[6,6],
 [7,7,7,7]]
Prelude Data.List> :i find
find :: (a -> Bool) -> [a] -> Maybe a -- Defined in
'Data.List'
Prelude Data.List> find (>4) [1,2,3,4,5,6]
Just 5
Prelude Data.List> find (>44) [1,2,3,4,5,6]
Nothing
```



elem, notElem, elemIndex, elemIndices

```
Prelude Data.List> elem 2 [1,2,3,4,5]
```

```
True
```

```
Prelude Data.List> notElem 2 [1,2,3,4,5]
```

```
False
```

```
Prelude Data.List> 4 `elemIndex` [1,2,3,4,5,6]
```

```
Just 3
```

```
Prelude Data.List> 10 `elemIndex` [1,2,3,4,5,6]
```

```
Nothing
```

```
Prelude Data.List> ' ' `elemIndices`  
"Facultatea de Informatica"
```

```
[10,13]
```

```
Prelude Data.List> ' ' `elemIndices`  
"FacultateadeInformatica"
```

```
[]
```

λ delete, \\, union, intersect, insert

```
Prelude Data.List> delete 6 [1,2,1,2,6,3,5,6]
[1,2,1,2,3,5,6]
Prelude Data.List> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
Prelude Data.List> [1,2,1,2,6,3,5,6]\\[6]
[1,2,1,2,3,5,6]
Prelude Data.List> [1,2,1,2,6,3,5,6]\\[6, 6]
[1,2,1,2,3,5]
Prelude Data.List> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude Data.List> [1..7] `intersect` [5..10]
[5,6,7]
Prelude Data.List> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
Prelude Data.List> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
Prelude Data.List> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
```



Data.Char

isControl , isSpace , isLower , isUpper , isAlpha , isAlphaNum ,
isPrint , isDigit , isOctDigit , isHexDigit , isLetter , isMark ,
isNumber , isPunctuation , isSymbol , isSeparator , isAscii ,
isLatin1 , isAsciiUpper , isAsciiLower, toUpper, toLower,
toTitle, digitToInt, intToDigit, ord, chr

```
Prelude> all isAlphaNum "bobby283"
<interactive>:2:5: Not in scope: `isAlphaNum'
Prelude> :m + Data.Char
Prelude Data.Char> all isAlphaNum "bobby283"
True
Prelude Data.Char> all isAlphaNum "eddy the fish!"
False
Prelude Data.Char> map ord "abcdefghijklmnopqrstuvwxyz"
[97,98,99,100,101,102,103,104]
Prelude Data.Char> chr 97
'a'
```



Data.Map

- **fromList** function takes an association list (in the form of a list) and returns a map with the same associations.

```
Prelude Map> Map.fromList [("betty","555-2938"),  
                           ("bonnie","452-2928"),("lucille","205-2928")]
```

```
fromList [("betty","555-2938"),  
          ("bonnie","452-2928"),("lucille","205-2928")]
```

- **empty**

```
Prelude Map> Map.empty
```

```
fromList []
```



insert, null, size

```
Prelude Map> let xm = Map.insert 3 100 Map.empty
Prelude Map> xm
fromList [(3,100)]
Prelude Map> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100
    Map.empty))
fromList [(3,100),(4,200),(5,600)]
Prelude Map> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100
    xm))
fromList [(3,100),(4,200),(5,600)]
Prelude Map> Map.null xm
False
Prelude Map> Map.size xm
1
Prelude Map> xm
fromList [(3,100)]
Prelude Map> let ym = Map.insert 5 600 (Map.insert 4 200 ( Map.insert
    3 100  xm))
Prelude Map> ym
fromList [(3,100),(4,200),(5,600)]
Prelude Map> Map.size ym
3
```



singleton, lookup, member, map, filter

```
Prelude Map> Map.singleton 3 9
fromList [(3,9)]
Prelude Map> Map.lookup 9 (Map.insert 5 9 $
  Map.singleton 3 9)
Nothing
Prelude Map> Map.lookup 3 (Map.insert 5 9 $
  Map.singleton 3 9)
Just 9
Prelude Map> Map.member 3 $ Map.fromList [(3,6),
  (4,3),(6,9)]
True
Prelude Map> Map.map (*100) $ Map.fromList [(1,1),
  (2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
Prelude Map Data.Char> Map.filter isUpper $ 
  Map.fromList [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```



toList, keys, elems, fromListWith, insertWith

```
Prelude Map Data.Char> Map.toList . Map.insert 9 2 $  
  Map.singleton 4 3  
[(4,3),(9,2)]  
Prelude Map Data.Char> Map.keys ym  
[3,4,5]  
Prelude Map Data.Char> Map.elems ym  
[100,200,600]  
Prelude Map Data.Char> Map.fromListWith max [(2,3),  
  (2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]  
fromList [(2,100),(3,29),(4,22)]  
Prelude Map Data.Char> Map.fromListWith (+) [(2,3),  
  (2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]  
fromList [(2,108),(3,62),(4,37)]  
Prelude Map Data.Char> Map.insertWith (+) 3 100 $  
  Map.fromList [(3,4),(5,103),(6,339)]  
fromList [(3,104),(5,103),(6,339)]
```



Data.Set

```
Prelude Set> let text = "Facultatea de  
Informatica"  
Prelude Set> let set1 = Set.fromList text  
Prelude Set> set1  
fromList " FIacdefilmnortu"  
Prelude Set> let set2 = Set.fromList  
[1,4,2,3,2,1,5,3,2,4,3]  
Prelude Set> set2  
fromList [1,2,3,4,5]  
Prelude Set> Set.fromList [True, True, False]  
fromList [False,True]  
Prelude Set> Set.fromList ["True", "True",  
"False"]  
fromList ["False","True"]
```



intersection, difference, union

```
Prelude Set> let set3 = Set.fromList  
[1,2,3,2,1,6,3,2,3,7]  
Prelude Set> set3  
fromList [1,2,3,6,7]  
Prelude Set> set2  
fromList [1,2,3,4,5]  
Prelude Set> Set.intersection set2 set3  
fromList [1,2,3]  
Prelude Set> Set.difference set2 set3  
fromList [4,5]  
Prelude Set> Set.difference set3 set2  
fromList [6,7]  
Prelude Set> Set.union set2 set3  
fromList [1,2,3,4,5,6,7]
```



null, size, member, empty, singleton, insert and delete

```
Prelude Set> Set.null Set.empty
```

```
True
```

```
Prelude Set> Set.null $ Set.fromList [3,4,5,5,4,3]
```

```
False
```

```
Prelude Set> Set.size $ Set.fromList [3,4,5,3,4,5]
```

```
3
```

```
Prelude Set> Set.singleton 9
```

```
fromList [9]
```

```
Prelude Set> Set.insert 4 $ Set.fromList [9,3,8,1]
```

```
fromList [1,3,4,8,9]
```

```
Prelude Set> Set.insert 8 $ Set.fromList [5..10]
```

```
fromList [5,6,7,8,9,10]
```

```
Prelude Set> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
```

```
fromList [3,5]
```



isSubsetOf, isProperSubsetOf, map, filter

```
Prelude Set> Set.fromList [2,3,4] `Set.isSubsetOf`  
  Set.fromList [1,2,3,4,5]
```

True

```
Prelude Set> Set.fromList [1,2,3,4,5]  
  `Set.isProperSubsetOf` Set.fromList [1,2,3,4,5]
```

False

```
Prelude Set> Set.fromList [2,3,4,8] `Set.isSubsetOf`  
  Set.fromList [1,2,3,4,5]
```

False

```
Prelude Set> Set.filter odd $ Set.fromList  
  [3,4,5,6,7,2,3,4]
```

```
fromList [3,5,7]
```

```
Prelude Set> Set.map (+1) $ Set.fromList  
  [3,4,5,6,7,2,3,4]
```

```
fromList [3,4,5,6,7,8]
```



Making our own modules

--Geometry.hs

```
module Geometry ( sphereVolume , sphereArea , cubeVolume , cubeArea , cuboidA  
rea , cuboidVolume ) where  
sphereVolume :: Float -> Float  
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)  
sphereArea :: Float -> Float  
sphereArea radius = 4 * pi * (radius ^ 2)  
cubeVolume :: Float -> Float  
cubeVolume side = cuboidVolume side side side  
cubeArea :: Float -> Float  
cubeArea side = cuboidArea side side side  
cuboidVolume :: Float -> Float -> Float -> Float  
cuboidVolume a b c = rectangleArea a b * c  
cuboidArea :: Float -> Float -> Float -> Float  
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2  
rectangleArea :: Float -> Float -> Float rectangleArea a b = a * b
```



Submodules: Sphere.hs, Cuboid.hs, Cube.hs

--Sphere.hs

```
module Geometry.Sphere ( volume , area )  
where  
volume :: Float -> Float  
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)  
area :: Float -> Float  
area radius = 4 * pi * (radius ^ 2)
```

```
import qualified Geometry.Sphere as Sphere  
import qualified Geometry.Cuboid as Cuboid  
import qualified Geometry.Cube as Cube
```



Cursul 9 – 10 Plan

- Tipul de dată abstract Set
 - Specificare algebrică
 - Implementare cu liste
 - Implementare cu arbori
 - Arboi echilibrați (AVL)
- Tipul abstract Bag
 - Reprezentare cu arbori heap



Mulțimi

- Mulțimile pot fi reprezentate prin:
 - Liste
 - Liste fără elemente duplicate
 - Liste ordonate
 - Arbori
 - Funcții booleene
 - etc.
- Reprezentarea este aleasă în funcție de operațiile ce se folosesc



Mulțimi

- Operații pe mulțimi:

empty :: Set a

isEmpty :: Set a -> Bool

member :: Set a -> a -> Bool

insert :: a -> Set a -> Set a

delete :: a -> Set a -> Set a

- Un tip bazat pe cele 5 operații: dicționar

union :: Set a -> Set a -> Set a

meet :: Set a -> Set a -> Set a

minus :: Set a -> Set a -> Set a



Multimi – specificarea algebrică

```
insert x (insert x xs) = insert x xs
```

```
insert x (insert y xs) = insert y (insert x xs)
```

```
isEmpty empty = True
```

```
isEmpty(insert x xs) = False
```

```
member empty y = False
```

```
member(insert x xs) y = (x == y) `or` member xs y
```

```
delete x empty = empty
```

```
delete x (insert y xs) = if x == y then delete x xs  
                           else insert y (delete x xs)
```



Multimi – specificarea algebrică

```
union xs empty = xs
```

```
union xs (insert y ys) = insert y (union xs ys)
```

```
meet xs empty = empty
```

```
meet xs (insert y ys) =
```

```
    if member xs y then insert y (meet xs ys)  
        else meet xs ys
```

```
minus xs empty = xs
```

```
minus xs (insert y ys) = minus (delete y xs) ys
```



Multimi – reprezentarea cu liste

- Presupunând că a este instanță a clasei Eq, multimile se pot reprezenta cu liste
- Funcția abstr:

```
abstr :: [a] -> Set a  
abstr = foldr insert empty
```

- Funcția de validare poate avea două variante:

```
valid xs = True

- În acest caz orice listă reprezintă o multime. Operațiile se implementează ușor dar sunt dezavantaje majore


```
valid xs = nonduplicated xs

- Se obțin rezultate mai bune la inserție și reuniune

```


```



Multimi – reprezentarea cu liste

- Dacă orice listă este o reprezentare validă atunci:

```
member xs x = some(==x) xs
insert x xs = x:xs
delete x xs = filter(\= x) xs
union xs ys = xs ++ ys
minus xs ys = filter(not.member ys) xs
```

```
some :: (a -> Bool) -> [a] -> Bool
some p = or.map p
```

- Avantaj: insert necesită timp constant
- Dezavantaj: lista poate fi mult mai mare decât multimea. Dacă n este lungimea reprezentării (listei), delete este de complexitate $\Theta(n)$ iar minus $\Theta(n^2)$, în timp ce multimea poate să aibă m elemente cu $m \ll n$.



Multimi – reprezentarea cu liste

- Dacă se restricționează reprezentarea la liste cu elemente distincte atunci, o primă variantă de implementare:

```
insert x xs = x:filter(\=x)xs
union xs ys = xs ++ filter(not.member xs)ys
```

- Dar complexitatea este mare: $\Theta(n)$ respectiv $\Theta(n^2)$. Dacă presupunem a instanță a lui Ord și impunem pentru validare liste strict ordonate atunci:

```
member xs x = if null xs then False
               else(x == head ys)
                   where ys = dropWhile (<x) xs
union [] ys = ys
union (x:xs) (y:ys) | (x<y) = x:union xs (y:ys)
                     | (x==y) = x:union xs ys
                     | (x>y) = y:union (x:xs) ys
```

λ Multimi – reprezentarea cu arbori

```
module Set(Set, empty, isEmpty, member, insert, delete) where

data Set a = Null | Fork (Set a) a (Set a)
deriving (Show)

empty :: Set a
empty = Null

isEmpty :: Set a -> Bool
isEmpty Null = True
isEmpty(Fork xt y zt) = False

member :: (Ord a) => Set a -> a -> Bool
member Null x = False
member (Fork xt y zt) x
| (x < y) = member xt x
| (x == y) = True
| (x > y) = member zt x
```

λ Multimi – reprezentarea cu arbori

```
insert :: (Ord a) => a -> Set a -> Set a

insert x Null = Fork Null x Null
insert x (Fork xt y zt)
  | (x < y)  = Fork(insert x xt) y zt
  | (x == y) = Fork xt y zt
  | (x > y)  = Fork xt y (insert x zt)

delete :: (Ord a) => a -> Set a -> Set a

delete x Null = Null
delete x (Fork xt y zt)
  | (x < y)  = Fork(delete x xt) y zt
  | (x == y) = join xt zt
  | (x > y)  = Fork xt y (delete x zt)

join :: Set a -> Set a -> Set a

join xt yt = if isEmpty yt then xt else Fork xt y zt
             where (y, zt) = splitTree yt

splitTree :: Set a -> (a, Set a)

splitTree(Fork xt y zt) = if isEmpty xt then (y, zt) else (u, Fork vt y zt)
                           where (u, vt) = splitTree xt
```

λ Multimi – reprezentarea cu arbori

- Funcțiile join și splitTree nu pot fi utilizate în scripturi ce importă modulul Set
- Funcția join nu poate fi folosită pentru reuniune: valoarea **join xt yt** este definită cu presupunerea că fiecare element din xt este mai mic decât fiecare element din yt
- Reprezentarea multimilor prin arbori este eficientă doar pentru operațiile dicționar: inserare, ștergere, căutare



Exemplu de utilizare

```
import Set

toSet :: Ord a => [a] -> Set a
toSet [] = empty
toSet (x:xs) = insert x (toSet xs)
t1, t2, t3 :: Set Int
t1 = toSet [4,1,6]
t2 = toSet [11,9,8]
t3 = empty

Main> toSet [8,3,5,2,1]
Fork Null 1 (Fork Null 2 (Fork (Fork Null 3 Null) 5 (Fork Null 8
Null)))

Main> insert 7 t1
Fork (Fork Null 1 (Fork Null 4 Null)) 6 (Fork Null 7 Null)
Main> insert 5 t2
Fork (Fork Null 5 Null) 8 (Fork Null 9 (Fork Null 11 Null))
Main> t2
Fork Null 8 (Fork Null 9 (Fork Null 11 Null))
Main> insert 10 t2
Fork Null 8 (Fork Null 9 (Fork (Fork Null 10 Null) 11 Null))
```



Mulțimi – reprezentarea cu arbori

- Probleme la reprezentarea cu arbori:
 - timpul de execuție pentru member, insert, delete depind de înălțimea arborelui
 - o mulțime cu n elemente poate fi reprezentată cu un arbore de înălțime $\log(n+1)$
 - pentru păstrarea relației $h = O(\log n)$ - arbori echilibrați (arbori AVL)



Arbore echilibrați (AVL)

- Introduși de G. Adelson-Velski și Y. Landis
- Un arbore este AVL dacă înălțimile subarborelor stâng respectiv drept pentru fiecare nod, diferă cu cel mult o unitate
- Formalizare: funcția **balanced**

```
balanced :: Stree a -> Bool
balanced Null = True
balanced(Fork xt x yt) = abs(height xt - height yt) <= 1
                           && balanced xt && balanced yt
```

- Arborele **xt** este echilibrat dacă
balanced xt este **True**



Arbore echilibrați (AVL)

Teoremă: Dacă x este un arbore echilibrat de dimensiune n și înălțimea h atunci

$$h \leq 1.4404\log(n+1) + O(1)$$

Consecință: Operațiile de bază pe arbori echilibrați (inserare, ștergere,...) sunt de complexitate $O(\log n)$

Implementarea operațiilor – aceeași schemă plus reechilibrare



Reprezentarea mulțimilor prin arbori AVL

- Folosim tipul Stree și construim un altul, ASTree, adăugând pentru fiecare arbore și valoarea înălțimii sale:

```
data ASTree a = Null  
              | Fork Int (ASTree a) a (ASTree a)
```

- Să considerăm o funcție ht care extrage dintr-un ASTree eticheta de tip Int. Atunci putem descrie o funcție fork ce păstrează invariantul: “Eticheta de tip Int a unui Astree xt reprezintă înălțimea lui xt”

```
ht :: ASTree a -> Int  
ht Null = 0  
ht(Fork h xt y zt) = h
```



Reprezentarea mulțimilor prin arbori AVL

- Funcția fork:

```
fork :: ASTree a -> a -> ASTree a -> ASTree a
fork xt y zt = Fork h xt y zt
    where h = 1 + (max (ht xt ht zt))
```

```
Main> fork Null 7 Null
Fork 1 Null 7 Null
Main> fork Null 7 (fork Null 2 (fork Null 3 Null))
Fork 3 Null 7 (Fork 2 Null 2 (Fork 1 Null 3 Null))
Main> fork Null 6 (Fork 4 Null 4 Null)
Fork 5 Null 6 (Fork 4 Null 4 Null)
```



Reprezentarea multimilor prin arbori AVL

- Funcția fork construiește arbore echilibrat numai dacă x_t și z_t sunt echilibrați și diferența de înălțime este cel mult 1.
- Pentru a asigura construcția numai a arborilor echilibrați va trebui să proiectăm o altă funcție, de aceelași tip:
`spoon :: ASTree a -> a -> ASTree a -> ASTree a`
- Funcțiile de inserare, stergere, etc. se definesc la fel ca în cazul Stree înlocuind Fork prin spoon



Funcțiile insert și delete

```
insert :: (Ord a) => a -> ASTree a -> ASTree a
insert x Null = fork Null x Null
insert x (Fork h xt y zt)
| (x < y)  = spoon (insert x xt) y zt
| (x == y) = Fork h xt y zt
| (x > y)  = spoon xt y (insert x zt)

delete :: (Ord a) => a -> ASTree a -> ASTree a
delete x Null = Null
delete x (Fork h xt y zt)
| (x < y)  = spoon(delete x xt) y zt
| (x == y) = join xt zt
| (x > y)  = spoon xt y (delete x zt)

join :: ASTree a -> ASTree a -> ASTree a
join xt yt = if isEmpty yt then xt else spoon xt y zt
             where (y, zt) = splitTree yt

splitTree :: ASTree a -> (a, ASTree a)
splitTree(Fork h xt y zt) = if isEmpty xt then (y,zt)
                           else (u,spoon vt y zt)
                           where(u,vt) = splitTree xt
```



Implementare spoon

- Fiecare inserare sau ștergere alterează înălțimea arborelui cu 1
- Este suficient să implementăm spoon $xt \ y \ zt$ în condițiile în care xt și zt sunt echilibrați și înălțimile lor diferă cu cel mult 2
- Dacă $ht\ xt = ht\ zt + 1$ atunci spoon este fork
- Să presupunem $ht\ xt = ht\ zt + 2$ și atunci xt este nevid:
$$xt = Fork\ n\ ut\ v\ wt$$
 - Cazul $ht\ zt = ht\ xt + 2$ este simetric
- spoon $xt \ y \ zt$ depinde de înălțimile relative ale lui ut și wt



Implementare spoon

- Cazul 1: $ht\ wt \leq ht\ ut$ Atunci:

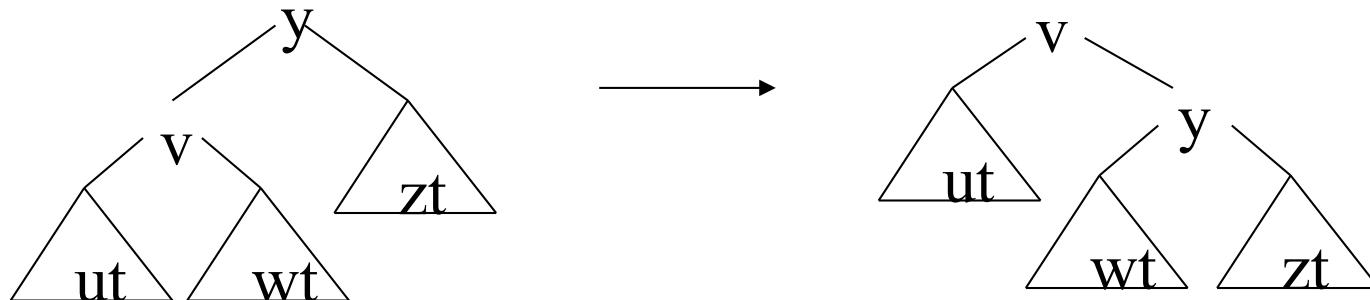
$$ht\ zt = ht\ xt - 2 = ht\ ut - 1 \leq ht\ wt \leq ht\ ut$$

În acest caz definim:

$$\text{spoon } xt\ y\ zt = \text{rotr}(\text{fork } xt\ y\ zt)$$

$$\text{rotr}(\text{Fork } m (\text{Fork } n \text{ ut } v \text{ wt}) \text{ y zt}) =$$

$$\text{fork } ut\ v (\text{fork } wt\ y\ zt)$$





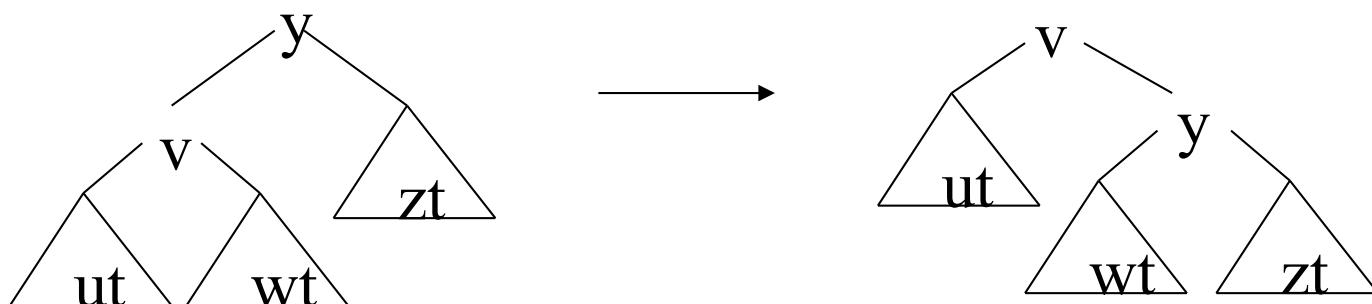
Implementare spoon

- Cazul 1 este corect:

$$\text{abs}(\text{ht ut} - \text{ht}(\text{fork wt y zt})) = \quad \quad \quad (\text{def ht})$$

$$\text{abs}(\text{ht ut} - 1 - \max(\text{ht wt}, \text{ht zt})) =$$

$$\text{abs}(\text{ht ut} - 1 - \text{ht wt}) \leq 1$$





Implementare spoon

- Cazul 2: $ht\ wt = 1 + ht\ ut$

Atunci wt nu poate fi vid; fie $wt = Fork\ p\ rt\ s\ tt$
iar $xt = Fork\ n\ ut\ v\ wt$

Au loc:

$$ht\ zt = ht\ xt - 2 = ht\ ut = ht\ wt - 1 = \max(ht\ rt, ht\ tt)$$

și atunci definim:

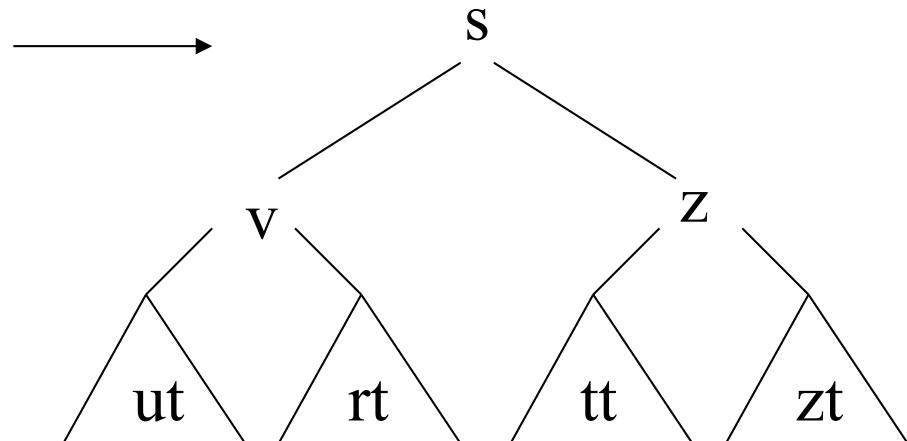
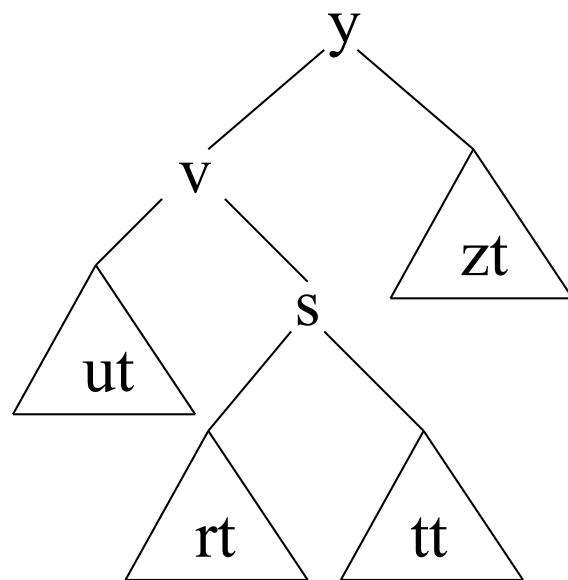
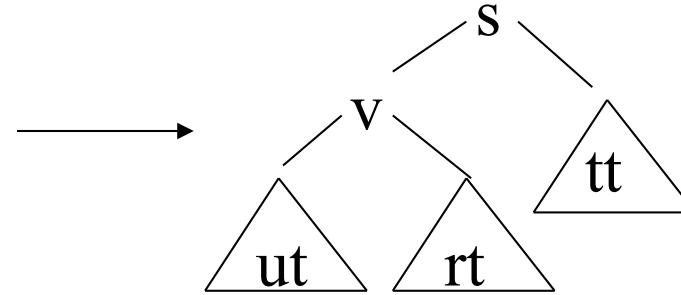
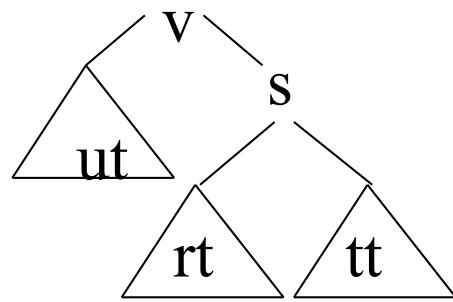
$$\text{spoon } xt\ y\ zt = \text{rotr}(\text{fork}(\text{rotl}\ xt)\ y\ zt)$$

unde

$$\text{rotl}(\text{Fork}\ n\ ut\ v\ (\text{Fork}\ p\ rt\ s\ tt)) = \text{fork}\ (\text{fork}\ ut\ v\ rt)\ s\ tt$$

λ

Rotăție dublă





Implementare spoon

- Cazul 2 este corect:

$\text{spoon } xt \ y \ zt = \text{fork}(\text{fork } ut \ v \ rt) \ s (\text{fork } tt \ y \ zt)$

și deci:

$$\text{abs(ht (fork ut v rt) - ht(fork tt y zt))} =$$

$$\text{abs(max(ht ut, ht rt) - max(ht tt, ht zt))} =$$

$$\text{abs(max(ht ut, ht rt) - ht zt)} =$$

$$\text{abs(ht ut - ht zt)} = 0$$



Implementare spoon

- Implementarea completă:

spoon :: ASTree a -> a -> ASTree a -> ASTree a

spoon xt y zt

(hz + 1 < hx) && (bias xt < 0)	= rotr(fork(rotl xt) y zt)
(hz + 1 < hx)	= rotr(fork xt y zt)
(hx + 1 < hz) && (0 < bias zt)	= rotl(fork xt y (rotr zt))
(hx + 1 < hz)	= rotl(fork xt y zt)
otherwise	= fork xt y zt

$$\begin{aligned} \text{where } \quad \text{hx} &= \text{ht xt} \\ \text{hz} &= \text{ht zt} \end{aligned}$$

bias :: ASTree a -> Int

bias(Fork h xt y zt) = ht xt - ht zt

rotr :: ASTree a -> ASTree a

rotr(Fork m (Fork n ut v wt) y zt) = fork ut v (fork wt y zt)

rotl :: ASTree a -> ASTree a

rotl(Fork m ut v (Fork n rt s tt)) = fork(fork ut v rt) s tt



Exemple

```
Main> insert 3 (insert 5 (insert 1 (insert 4
  (insert 2 Null))))
Fork 3 (Fork 1 Null 1 Null) 2 (Fork 2 (Fork 1 Null
  3 Null) 4 (Fork 1 Null 5 Null
))
Main> insert 3 (insert 5 (insert 1 (insert 4
  (insert 2 (insert 0 Null)))))

Fork 3 (Fork 2 Null 0 (Fork 1 Null 1 Null)) 2 (Fork
  2 (Fork 1 Null 3 Null) 4 (Fork 1 Null 5 Null))
Main> insert 3 (insert 5 (insert 1 (insert 4
  (insert 2 (insert 10 Null)))))

Fork 3 (Fork 2 (Fork 1 Null 1 Null) 2 (Fork 1 Null
  3 Null)) 4 (Fork 2 (Fork 1 Null 5 Null) 10 Null)
```



Multiset-uri (Bags)

- Nu contează ordinea elementelor
 $\{1,2,2,3\} = \{3,2,1,2\}$
- Contează elementele duplicate
 $\{1,2,2,3\} \neq \{1,2,3\}$
- Tipul Bag a cu a din clasa Ord
- Operări:
 - mkBag :: [a] -> Bag a
 - isEmpty :: Bag a -> Bool
 - union :: Bag a -> Bag a -> Bag a
 - minBag :: Bag a -> a
 - delMin :: Bag a -> Bag a



Multiset-uri (Bags)

- Specificare algebrică:
 - $\text{isEmpty}(\text{mkBag } xs) = \text{null } xs$
 - $\text{union}(\text{mkBag } xs) (\text{mkBag } ys) = \text{mkBag}(xs ++ ys)$
 - $\text{minBag}(\text{mkBag } xs) = \text{minlist } xs$
 - $\text{delMin}(\text{mkBag } xs) = \text{mkBag}(\text{deleteMin } xs)$

unde deleteMin este funcția ce elimină o apariție a celui mai mic element dintr-o listă



Multiset-uri (Bags)

- Implementare prin liste nedescrescătoare
 - union este implementată cu *merge*
 - mkBag este implementat cu *sort*
- Complexitatea operațiilor:

– mkBag xb	$O(n \log n)$
– isEmpty xb	$O(1)$
– union xb yb	$O(m + n)$
– minBag xb	$O(1)$
– delMin xb	$O(1)$



Multiset-uri (Bags)

- Implementare prin arbori heap augmentați
- Complexitatea operațiilor:
 - mkBag xb O(n)
 - isEmpty xb O(1)
 - union xb yb O(log m + log n)
 - minBag xb O(1)
 - delMin xb O(log n)



Multiset-uri (Bags)

- Arbori heap augmentați
 - Arbori heap introduși anterior:

```
data (Ord a) => Htree a = Null | Fork a (Htree a) (Htree a)
```

- Se adaugă un întreg – dimensiunea subarborelui stâng

```
data Htree a = Null | Fork Int a (Htree a) (Htree a)
```

- Arbore “leftist”: Dimensiunea fiecărui subarbore stâng este cel puțin cât și dimensiunea subarborelui drept corespunzător



Multiset-uri (Bags)

- Construirea cu funcția fork:
 - Dintr-o etichetă și 2 arbori heap face un heap la care se adaugă informația dimensiune și, eventual, schimbă ordinea membrilor

```
fork :: (Ord a) => a -> Htree a -> Htree a -> Htree a
fork x yt zt = if m < n then Fork p x zt yt else Fork p x yt zt
                where   m = size yt
                      n = size zt
                      p = m + n + 1

size :: Ord a=>Htree a -> Int
size Null = 0
size(Fork n x yt zt) = n
```



Multiset-uri (Bags)

- Implementarea operațiilor:

```
isEmpty :: Htree a -> Bool  
isEmpty Null = True  
isEmpty(Fork n x yt zt) = False
```

```
minBag :: Htree a -> a  
minBag(Fork n x yt zt) = x
```

```
delMin :: Htree a -> Htree a  
delMin(Fork n x yt zt) = union yt zt
```



Multiset-uri (Bags)

- Implementarea operațiilor:

```
union :: Htree a -> Htree a -> Htree a
union Null yt = yt
union(Fork m u vt wt) Null = Fork m u vt wt
union(Fork m u vt wt) (Fork n x yt zt)
    | u <= x = fork u vt (union wt (Fork n x yt zt))
    | x < u  = fork x yt (union (Fork m u vt wt) zt)
```



Multiset-uri (Bags)

- Implementarea operațiilor:

```
mkBag :: [a] -> Htree a
```

```
mkBags xs = fst(mkTwo(length xs) xs)
```

```
mkTwo :: Int -> [a] -> (Htree a , [a])
```

```
mkTwo n xs
```

```
| (n == 0) = (Null, xs)
```

```
| (n == 1) = (fork(head xs) Null Null , tail xs)
```

```
| otherwise= (union xt yt, zs)
```

```
where (xt, ys) = mkTwo m xs
```

```
      (yt, zs) = mkTwo(n-m) ys
```

```
      m           = n div 2
```



Multiset-uri (Bags) - Exemple

```
Main> mkBag [8,4,2,2,3,2,1]
Fork 7 1 (Fork 4 2 (Fork 2 2 (Fork 1 8 Null Null) Null) (Fork 1 4
    Null Null)) (Fork 2 2 (Fork 1 3 Null Null) Null)
Main> mkBag [2,2,2,3]
Fork 4 2 (Fork 2 2 (Fork 1 3 Null Null) Null) (Fork 1 2 Null Null)

Main> delMin(mkBag [2,2,2,3])
Fork 3 2 (Fork 1 3 Null Null) (Fork 1 2 Null Null)

Main> minBag(mkBag [2,2,2,3])
2
Main> union (mkBag [2,2,2,3]) (mkBag [2,3,1,1])
Fork 8 1 (Fork 5 1 (Fork 4 2 (Fork 2 2 (Fork 1 3 Null Null) Null)
    (Fork 1 2 Null Null)) Null) (Fork 2 2 (Fork 1 3 Null Null) Null)

Main> delMin (union (mkBag [2,2,2,3]) (mkBag [2,3,1,1]))
Fork 7 1 (Fork 4 2 (Fork 2 2 (Fork 1 3 Null Null) Null) (Fork 1 2
    Null Null)) (Fork 2 2 (Fork 1 3 Null Null) Null)
```



Cursul 11 – 12 MONADE

- Monade
 - Introducere
 - Tipul de data Maybe
 - Constructori de tip
 - Monada în Haskell
 - Exemplul 1: monada Maybe
 - Exemplul 2: monada []
 - Clasa Monad
 - Notația do
 - Monada IO
 - Exemple



“Side Effects” în Haskell

- Programele de până acum nu au “side-effects”: programele sunt executate pentru a afla niște valori care se afișează.
- Cum construim programe interactive? (citirea unor valori, afișare, lucru cu fișiere, desenarea unei figuri etc.)
- În Haskell, “valorile pure” sunt separate de “acțiuni”, în două moduri:
 - **Tipuri:** O expresie de tip `IO a` are asociate execuției sale eventuale *acțiuni*, ultima dintre ele fiind returnare unei valori de tip `a`.
 - **Sintaxă:** Construcția sintactică `do` realizează o acțiune care, în general, este secvență de acțiuni “elementare”.



Introducere

- **Monada** – o cale de a structura calculele ca valori și secvențe de calcule ce utilizează aceste valori
 - Valorile monadice sunt numite calcule
- Permit construirea de calcule folosind blocuri secvențiale care, la rândul lor pot fi secvențe de calcul
- Monada determină **modul în care o combinație de calcule formează un nou calcul**, scutindu-l pe programator de a scrie cod pentru astfel de combinații
- O monadă este văzută ca o strategie pentru combinarea calculelor într-un calcul complex



Tipul de data Maybe

- Tipul “**calcul care poate eşua sau poate returna o valoare**”:
data Maybe a = Nothing | Just a

```
mydiv::: Float -> Float -> Maybe Float
mydiv x 0 = Nothing
mydiv x y = Just (x/y)
```

- Tipul Maybe sugerează o strategie de combinare a calculelor: dacă un calcul compus constă dintr-un calcul B care depinde de rezultatul altui calcul A, atunci calculul combinat va produce Nothing dacă A sau B produc Nothing; altfel, dacă ambele se produc cu succes, calculul combinat va produce rezultatul calculului B, calcul care este aplicat rezultatului calculului A



Proprietăți fundamentale

- **Modularitate**
 - calcule (complexe) compuse din calcule simple
 - separarea strategiilor de combinare a calculelor de calculele în sine
- **Flexibilitate**
 - Programele scrise cu monade sunt mai adaptabile decât cele fără monade
- **Izolare**
 - monadele permit crearea de structuri de calcul în stil imperativ (sistemul I/O de ex.) care rămân izolate de corpul principal al programului funcțional



Constructori de tip

- Un **constructor de tip** este o definiție de tip parametrizat ce folosește tipuri polimorfe
- În definiția tipului Maybe

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

Maybe este un constructor de tip iar **Nothing**, **Just** sunt **constructori de date** (de tipul Maybe)

- Se poate construi o data aplicând constructorul de data **Just** unei valori:

```
tara = Just "Romania"
```

- Se poate construi un tip aplicând constructorul de tip **Maybe** unui tip:

```
Maybe Int, Maybe String
```



- Tipurile polimorfe pot fi privite ca niște **containere** ce sunt capabile să conțină valori de diverse tipuri:
 - **Maybe String** ar putea fi privit ca un container **Maybe** ce poate conține o valoare de tip **String** (sau **Nothing**)
- Se poate ca tipul containerului să fie polimorf: **m a** reprezintă un **container de un anumit tip m** ce poate conține **o valoare de un anumit tip a**
- Se folosesc adesea variabile tip cu constructorii de tip pentru a descrie trăsăturile abstracte ale unui calcul: **Maybe a** este tipul tuturor calculelor care pot returna o valoare de tip a sau **Nothing**



Monada în Haskell

- O monadă în Haskell este dată prin:
 - un constructor de tip (notat m) care este numit **constructorul de tip monadă**
 - O funcție – constructor de tip, $a \rightarrow m\ a$, care construiește instanțe ale monadei, numită **return**
 - O funcție $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ care combină o instanță $m\ a$ cu un calcul (funcție) ce produce dintr-o valoare de tip a o instanță $m\ b$ a monadei și în final produce o nouă instanță $m\ b$. Această funcție este denumită **bind** și se scrie **>>=**



Monada în Haskell

- constructorul de tip monadă definește un tip de calcul
- funcția **return** creează valori primitive (instante) ale acestui tip
- operatorul (funcția) **>>=** combină calcule de acest tip pentru a obține calcule mai complexe de tipul respectiv
- Analogia cu containerul:
 - constructorul de tip **m** este un **container** ce poate conține **diferite valori**
 - **m a** este un **container** ce conține **o valoare de tip a**
 - funcția **return** pune **o valoare** în containerul monadă
 - funcția **>>=**:
 - ia valoarea din container
 - o transmite unei funcții
 - produce un nou container ce conține o nouă valoare, posibil de alt tip
 - funcția **>>=** se cheamă **bind** pentru că ea leagă valoarea din containerul monadă cu primul argument al funcției



Acces la componentele unei date

- Tipul de dată Point:

```
data Point = Pt Float Float  
let p = Pt 3.2 5.4
```

- Accesul la componente poate fi făcut prin funcții:

```
pointx :: Point -> Float  
pointx (Pt x _) = x  
pointx p
```

- Mai comod declarația echivalentă:

```
data Point = Pt {pointx, pointy :: Float}  
  
pointx :: Point -> Float  
pointy :: Point -> Float
```



Acces la componentele unei date

```
data T      = C1 {f1,f2 :: Int}  
             | C2 {f1 :: Int,  
                   f3,f4 :: Char}
```

Expresie

C1 {f1 = 3}

C2 {f1 = 1, f4 = 'A', f3 = 'B'}

x {f1 = 1}

Traducere

C1 3 undefined

C2 1 'B' 'A'

```
case x of C1 _ f2    -> C1 1 f2  
           C2 _ f3 f4 -> C2 1 f3 f4
```



Exemplu: Monada **Maybe**

- Studiu de caz: determinarea strămoșilor: bunci, străbunici etc. din arborele genealogic
- Tipul de date Person:

```
data Person = Person {name :: String,  
                      mother :: Maybe Person, father :: Maybe Person}
```

- Pentru a determina buncul unei persoane am putea defini o funcție:

```
maternalGrandfather :: Person -> Maybe Person  
maternalGrandfather s = case (mother s) of  
                           Nothing -> Nothing  
                           Just m -> father m
```



Exemplu: Monada **Maybe**

- Un alt strămoș:

```
mothersPaternalGrandfather :: Person -> Maybe Person
mothersPaternalGrandfather s = case (mother s) of
                                Nothing -> Nothing
                                Just m -> case (father m) of
                                              Nothing -> Nothing
                                              Just gf -> father gf
```

- Soluția nu este eficientă
- Odată găsită valoarea **Nothing** într-un punct al calculului, aceasta rămâne **Nothing** ca rezultat final
- Sa încercăm să implementăm acest lucru o singură dată



Exemplu: Monada **Maybe**

- Crearea unui combinator ce înglobează acest aspect:

```
comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing = Nothing
comb (Just x) f = f x
```

- Utilizarea acestui combinator pentru a construi secvențe mai complicate:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather s = (Just s) `comb` mother `comb` father
```

```
fathersMaternalGrandmother :: Person -> Maybe Person
fathersMaternalGrandmother s = (Just s) `comb` father `comb`
                                mother `comb` mother
```

```
mothersPaternalGrandfather :: Person -> Maybe Person
mothersPaternalGrandfather s = (Just s) `comb` mother `comb`
                                father `comb` father
```



Exemplu: Monada **Maybe**

- Combinatorul **comb**
 - Este o funcție polimorfă, nu este specializată pentru tipul Person
 - Capturează strategia generală de combinare a calculelor care pot, în particular, să eșueze
 - Poate fi folosit și în alte aplicații: interogare baze de date, căutare în dicționare, etc.
- Tipul **Maybe** împreună cu funcția **Just** (care acționează ca și **return**) și combinatorul **comb** (ce acționează ca și **>>=**) formează o monadă folosită pentru construirea de calcule ce pot eșua



Monada []

- Constructorul de tip [] (pentru construirea listelor) este de asemenea o monadă:
 - Funcția return crează lista singleton

```
return x = [x]
```
 - Operația de legare pentru liste construește o nouă listă conținând rezultatul aplicării funcției tuturor valorilor listei originale

```
l >>= f = concatMap f l
concatMap :: (a -> [b]) -> [a] -> [b]
```



Clasa **Monad**

- În Haskell există o clasă standard **Monad** care definește numele și signatura funcțiilor **return** și **>>=**

```
class Monad m where
    (=>)   :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

- Se recomandă ca monadele utilizator să fie instanțe ale clasei **Monad**



Exemplul 1 revizuit

- Monada Maybe este instanță a clasei Monad:

```
instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return = Just
```

- Se pot utiliza operatorii standard din Monad:

```
maternalGrandfather s = (return s) >>= mother >>= father
```

```
fathersMaternalGrandmother s = (return s) >>= father >>=
mother >>= mother
```



Notația do

- Alternativa în a utiliza funcțiile monadice
- Similară cu utilizarea “list comprehensions” pentru liste
- Scrierea calculelor monadice în stil pseudo-imperativ folosind variabile:
 - Rezultatul unui calcul monadic poate fi “asignat” unei variabile folosind operatorul `<-`
 - Utilizarea variabilei într-o subsecvență de calcul monadic, se face automat legarea
 - Tipul expresiei din dreapta semnului `<-` este tip monadic



Notatia do

```
mothersPaternalGrandfather s = do m <- mother s  
                                  gf <- father m  
                                  father gf
```

sau:

```
mothersPaternalGrandfather s =  
    do {m <- mother s; gf <- father m; father gf}
```

- Monadele oferă prin notația do posibilitatea de a crea calcule în stil imperativ în cadrul programelor funcționale
- Notația do este doar “syntactic sugar”



Transformarea do în >>=

- Comanda **x <- expr1** devine:

expr1 >>= \x ->

- Comanda **expr2** devine:

expr2 >>= _ ->

- De exemplu:

```
mothersPaternalGrandfather s = do m <- mother s  
                                  gf <- father m  
                                  father gf
```

devine

```
mothersPaternalGrandfather s = mother s >>= \m ->  
                               father m >>= \gf ->  
                               father gf
```

- Operatorul **>>=** (bind) leagă valoarea din monadă cu argumentul din următoarea lambda expresie



Monada IO

- Operațiile de intrare/ieșire sunt incompatibile cu stilul funcțional pur: efect secundar, modificarea valorilor
- Soluția: monada IO a
 - Calcule ce realizează I/O în cadrul monadei IO
 - Calculele în monada IO se numesc acțiuni de intrare /ieșire (*I/O actions*)



“Side Effects” în Haskell

- Un program interactiv este conceput ca o funcție care are ca argument “state of the world” (o valoare de tip `World`) și produce o valoare de tip `World` care este o altă stare ce reflectă efectul colateral produs de program:

```
type IO = World -> World
```

- În general, un program interactiv poate returna și o anume valoare pe lângă efectul ce-l are asupra stării (de exemplu, se citește un caracter de la tastatură (acțiune) și se returnează acel caracter). Așadar, mai corect:

```
type IO a = World -> (a, World)
```

Exemplu: `IO Char`, `IO Int`; `IO ()`

- O expresie de tip `IO a` are asociate execuției sale *acțiuni*, ultima dintre ele fiind returnarea unei valori de tip `a`.



Construcția sintactică do

- Dacă **act** este o acțiune de tip **IO a** atunci putem descrie realizarea acțiunii **act**, returna valoarea corespunzătoare și eventual adăugarea altor acțiuni, astfel:

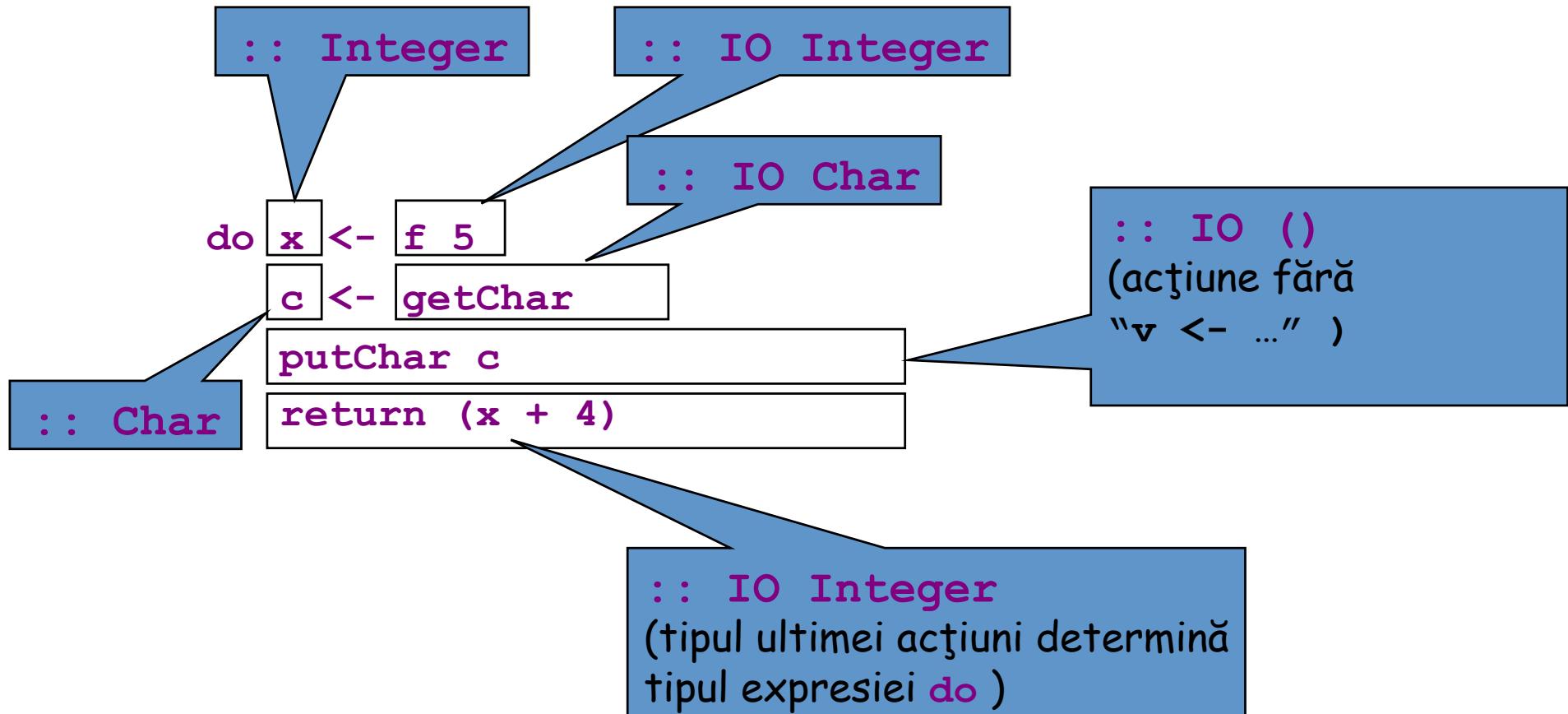
```
do val <- act
    ...
    ...
    return x
```

-- actiunea urmatoare
-- actiunea urmatoare
-- actiunea finală

- Toate acțiunile de după **val <- act** pot folosi **val**.
- Funcția **return** are un parametru de tip **a**, devine o acțiune de tip **IO a**, care nu face altceva decât să returneze valoarea respectivă.

λ

do Exemplu





Acțiuni IO

- O valoare de tip `IO a` este o acțiune; această valoare va avea efect atunci când va fi executată.
- În Haskell, o valoare program este valoarea variabilei `main` din modulul `Main`. Dacă această valoare este de tip `IO a`, atunci va fi executată ca o acțiune. Dacă este de alt tip atunci valoarea sa va fi afișată.



Actiuni IO predefinite

- Introducerea unui caracter de la tastatură

getChar :: IO Char

- Scrie un caracter la terminal

putChar :: Char -> IO ()

- Introducerea unei linii de la tastatură

getLine :: IO String

- Citirea unui fișier ca și String

readFile :: FilePath -> IO String

- Scriere String în fișier

writeFile :: FilePath -> String -> IO ()



Acțiuni recursive

- Acțiunea `getLine` poate fi definită recursiv din acțiuni mai simple:

```
getLine :: IO String
getLine = do c <- getChar          -- citește un caracter
            if c == '\n'        -- dacă este newline
                then return []  -- returnăm sirul vid
                else do l <- getLine -- recursiv restul
                          return (c:l) -- returnăm întreaga linie
```



Acțiuni recursive

- Acțiunea **putStr** poate fi definită recursiv din acțiuni mai simple:

```
putStr :: String -> IO ()
```

```
putStr []      = return ()  
putStr (x:xs) = do putChar x  
                  putStr xs
```

```
putStrLn :: String -> IO ()
```

```
putStrLn xs   = do putStr xs  
                  putChar '\n'
```



Exemplul 1: Lungimea unui sir citit

- Acțiunea de citire a unui sir de la tastatură și afișarea lungimii sale:

```
strlen:: IO()
strlen = do putStrLn "Introdu un sir: "
           xs <- getLine
           putStrLn "Sirul are "
           putStrLn (show (length xs))
           putStrLn " caractere"
```

Main> strlen

Introdu un sir: abracadabra

Sirul are 11 caractere



Exemplul 2: Suma unor intregi

```
getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)

sumInts :: IO Int
sumInts = do n <- getInt
             if n==0
                 then return 0
             else
                 do m <- sumInts
                     return (n+m)

suma::IO ()
suma = do x <- sumInts
          putStrLn("Suma este: " ++ show x )
```

λ

Main> suma

0

Suma este: 0

Main> suma

354

24

100

0

Suma este: 478



Exemplul 3: Comanda Unix wc

- Programul Unix wc (word count) citește un fișier și determină numărul de caractere, cuvinte și linii pe care le afișează
- Citirea fișierului este o acțiune, restul este un calcul.
- Strategia:
 - Se definește o funcție care determină numărul de caractere, cuvinte și linii într-un string.
 - Numărul de linii = numărul de '\n'
 - Numărul de cuvinte ~= numărul de ' ' plus numărul de '\t'
 - Se definește o acțiune care citește un fișier într-un string, aplică funcția, apoi afișează rezultatul.



Implementarea

```
wcf :: (Int,Int,Int) -> String -> (Int,Int,Int)
wcf (cc,w,lc) []           = (cc,w,lc)
wcf (cc,w,lc) (' ' : xs) = wcf (cc+1,w+1,lc) xs
wcf (cc,w,lc) ('\t' : xs) = wcf (cc+1,w+1,lc) xs
wcf (cc,w,lc) ('\n' : xs) = wcf (cc+1,w+1,lc+1) xs
wcf (cc,w,lc) (x : xs)    = wcf (cc+1,w,lc) xs

wc :: IO ()
wc = do putStrLn "Dati numele fisierului: "
        name      <- getLine
        contents <- readFile name
        let (cc,w,lc) = wcf (0,0,0) contents
            putStrLn ("Fisierul: " ++ name ++ " are ")
            putStrLn (show cc ++ " caractere ")
            putStrLn (show w ++ " cuvinte ")
            putStrLn (show lc ++ " linii ")
```



Exemplu execuție

*Main> wc

Dati numele fisierului: wcount.hs

Fisierul: wcount.hs are

615 caractere

173 cuvinte

17 linii

Introducere în teoria categoriilor

Categories in theory

1.1 Definition A **category** $\mathcal{C} = (\mathcal{O}_{\mathcal{C}}, \mathcal{M}_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}}, \circ_{\mathcal{C}}, \text{Id}_{\mathcal{C}})$ is a structure consisting of *morphisms* $\mathcal{M}_{\mathcal{C}}$, *objects* $\mathcal{O}_{\mathcal{C}}$, a *composition* $\circ_{\mathcal{C}}$ of morphisms, and *type information* $\mathcal{T}_{\mathcal{C}}$ of the morphisms, which obey to the following constraints.

Note, that we often omit the subscription with \mathcal{C} if the category is clear from the context.

1. We assume the collection of **objects** \mathcal{O} to be a set. Quite often the objects themselves also are sets, however, category theory makes *no* assumption about that, and provides no means to explore their structure.
2. The collection of **morphisms** \mathcal{M} is also assumed to be a set in this guide.
3. The ternary relation $\mathcal{T} \subseteq \mathcal{M} \times \mathcal{O} \times \mathcal{O}$ is called the **type information** of \mathcal{C} . We want every morphism to have a type, so a category requires

$$\forall f \in \mathcal{M} \quad \exists A, B \in \mathcal{O} \quad (f, A, B) \in \mathcal{T} .$$

We write $f : A \xrightarrow{\mathcal{C}} B$ for $(f, A, B) \in \mathcal{T}_{\mathcal{C}}$. Also, we want the types to be unique, leading to the claim

$$f : A \rightarrow B \wedge f : A' \rightarrow B' \Rightarrow A = A' \wedge B = B' .$$

This gives a notion of having a morphism to “map from one object to another”. The uniqueness entitles us to give names to the objects involved in a morphism. For $f : A \rightarrow B$ we call $\text{src } f := A$ the **source**, and $\text{tgt } f := B$ the **target** of f .

4. The *partial* function $\circ : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$, written as a binary infix operator, is called the **composition** of \mathcal{C} . Alternative notations are $f ; g := gf := g \circ f$.

Categories in theory

Morphisms can be composed whenever the target of the first equals the source of the second. Then the resulting morphism maps from the source of the first to the target of the second:

$$f : A \rightarrow B \wedge g : B \rightarrow C \Rightarrow g \circ f : A \rightarrow C$$

In the following, the notation $g \circ f$ implies these type constraints to be fulfilled.

Composition is associative, i.e. $f \circ (g \circ h) = (f \circ g) \circ h$.

5. Each object $A \in \mathcal{O}$ has associated a unique **identity morphism** id_A . This is denoted by defining the function

$$\begin{aligned} \text{Id} &: \mathcal{O} \longrightarrow \mathcal{M} \\ A &\longmapsto \text{id}_A , \end{aligned}$$

which automatically implies uniqueness.

The typing of the identity morphisms adheres to

$$\forall A \in \mathcal{O} \quad \text{id}_A : A \rightarrow A .$$

To deserve their name, the identity morphisms are —depending on the types— left or right neutral with respect to composition, i.e.,

$$f \circ \text{id}_{\text{src } f} = f = \text{id}_{\text{tgt } f} \circ f .$$

Spot a category in Haskell

With the last section in mind, where would you look for “the obvious” category? It is not required that it models the whole Haskell language, instead it is enough to point at the things in Haskell that behave like the objects and morphisms of a category.

We call our Haskell category \mathcal{H} and use Haskell’s types—primitive as well as constructed—as the objects $\mathcal{O}_{\mathcal{H}}$ of the category. Then, unary Haskell functions correspond to the morphisms $\mathcal{M}_{\mathcal{H}}$, with function signatures of unary functions corresponding to the type information $\mathcal{T}_{\mathcal{H}}$.

$$f :: A \rightarrow B \text{ corresponds to } f : A \xrightarrow{\mathcal{H}} B$$

Haskell’s function composition ‘.’ corresponds to the composition of morphisms $\circ_{\mathcal{H}}$. The identity in Haskell is typed

$$\text{id} :: \text{forall } a. a \rightarrow a$$

corresponding to

$$\forall A \in \mathcal{O}_{\mathcal{H}} \quad \text{id}_A : A \rightarrow A .$$

Note that we do not talk about n -ary functions for $n \neq 1$. You can consider a function like $(+)$ to map a number to a function that adds this number, a technique called **currying** which is widely used by Haskell programmers. Within the context of this guide, we do not treat the resulting function as a morphism, but as an object.

Functors in theory

4.1.1 Definition Let \mathcal{A}, \mathcal{B} be categories. Then two mappings

$$F_{\mathcal{O}} : \mathcal{O}_{\mathcal{A}} \rightarrow \mathcal{O}_{\mathcal{B}} \quad \text{and} \quad F_{\mathcal{M}} : \mathcal{M}_{\mathcal{A}} \rightarrow \mathcal{M}_{\mathcal{B}}$$

together form a **functor** F from \mathcal{A} to \mathcal{B} , written $F : \mathcal{A} \rightarrow \mathcal{B}$, iff

1. they preserve type information, i.e.,

$$\forall f : A \xrightarrow[\mathcal{A}]{} B \quad F_{\mathcal{M}} f : F_{\mathcal{O}} A \xrightarrow[\mathcal{B}]{} F_{\mathcal{O}} B ,$$

2. $F_{\mathcal{M}}$ maps identities to identities, i.e.,

$$\forall A \in \mathcal{O}_{\mathcal{A}} \quad F_{\mathcal{M}} \text{id}_A = \text{id}_{F_{\mathcal{O}} A} ,$$

3. and application of $F_{\mathcal{M}}$ distributes under composition of morphisms, i.e.,

$$\forall f : A \xrightarrow[\mathcal{A}]{} B ; g : B \xrightarrow[\mathcal{A}]{} C \quad F_{\mathcal{M}}(g \circ_{\mathcal{A}} f) = F_{\mathcal{M}} g \circ_{\mathcal{B}} F_{\mathcal{M}} f .$$

4.1.2 Notation Unless it is required to refer to only one of the mappings, the subscripts \mathcal{M} and \mathcal{O} are usually omitted. It is clear from the context whether an object or a morphism is mapped by the functor.

4.1.3 Definition Let \mathcal{A} be a category. A functor $F : \mathcal{A} \rightarrow \mathcal{A}$ is called **endofunctor**.

Functors in Haskell

Following our idea of a Haskell category \mathcal{H} , we can define an endofunctor $F : \mathcal{H} \rightarrow \mathcal{H}$ by giving a unary type constructor F , and a function $fmap$, constituting $F_{\mathcal{O}}$ and $F_{\mathcal{M}}$ respectively.

The type constructor F is used to construct new types from existing ones. This action corresponds to mapping objects to objects in the \mathcal{H} category. The definition of F shows how a functor implements the structure of the constructed type. The function $fmap$ lifts each function with a signature $f : A \rightarrow B$ to a function with signature $Ff : FA \rightarrow FB$.

Hence, functor application on an object is a *type level* operation in Haskell, while functor application on a morphism is a *value level* operation.

Haskell comes with the definition of a class

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

which allows overloading of $fmap$ for each functor.

Exemple: Maybe, []

The mapping function `fmap` differs from functor to functor. For the `Maybe` structure, it can be written as

```
fmap :: (a -> b) -> Maybe a -> Maybe b  
fmap f Nothing = Nothing  
fmap f (Just x) = Just (f x)
```

while `fmap` for the `List` structure can be written as

```
fmap :: (a -> b) -> [a] -> [b]  
fmap f [] = []  
fmap f (x:xs) = (f x):(fmap f xs)
```

Note, however, that having a type being a member of the Functor class does not imply to have a functor at all. Haskell does not check validity of the functor properties, so we have to do it by hand:

1. $f : A \rightarrow B \Rightarrow Ff : FA \rightarrow FB$ is fulfilled, since being a member of the Functor class implies that a function `fmap` with the according signature is defined.
2. $\forall A \in \mathcal{O} \quad F\text{id}_A = \text{id}_{FA}$ translates to

```
fmap id == id
```

which must be checked for each type one adds to the class. Note, that Haskell overloads the `id` function: The left occurrence corresponds to id_A , while the right one corresponds to id_{FA} .

3. $F(g \circ f) = Fg \circ Ff$ translates to

```
fmap (g . f) == fmap g . fmap f
```

which has to be checked, generalising over all functions `g` and `f` of appropriate type.

Maybe and List are both functors.

For the Maybe structure, the data constructors are `Just` and `Nothing`. So we prove the second functor property, $F \text{id}_A = \text{id}_{FA}$, by

```
fmap id Nothing
  == Nothing
  == id Nothing

fmap id (Just y)
  == Just (id y)
  == Just y
  == id (Just y) ,
```

and the third property, $F(g \circ f) = Fg \circ Ff$, by

```
(fmap g . fmap h) Nothing
  == fmap g (fmap h Nothing)
  == fmap g Nothing
  == Nothing
  == fmap (g . h) Nothing ,

(fmap g . fmap h) (Just y)
  == fmap g (fmap h (Just y))
  == fmap g (Just (h y))
  == Just (g (h y))
  == Just ((g . h) y)
  == fmap (g . h) (Just y) .
```

Maybe and List are both functors.

Note, that List is—in contrast to Maybe—a recursive structure. This can be observed in the according definition of `fmap` above, and it urges us to use induction in the proof: The second property, $F \text{id}_A = \text{id}_{FA}$, is shown by

```
fmap id []
== []
== id []

fmap id (x:xs)
== (id x) : (fmap id xs)
== (id x) : (id xs)  --here we use induction
== x:xs
== id (x:xs) ,
```

and the third property, $F(g \circ f) = Fg \circ Ff$ can be observed in

```
fmap (g . f) []
== []
== fmap g []
== fmap g (fmap f [])
== (fmap g . fmap f) []
```

Maybe and List are both functors.

```
fmap (g . f) (x:xs)
== ((g . f) x):(fmap (g . f) xs)
== ((g . f) x):((fmap g . fmap f) xs)    -- induction
== (g (f x)):(fmap g (fmap f xs))
== fmap g ((f x):(fmap f xs))
== fmap g (fmap f xs)
== (fmap g . fmap f) xs .
```

Natural transformations in theory

5.1.1 Definition Let \mathcal{A}, \mathcal{B} be categories, $F, G : \mathcal{A} \rightarrow \mathcal{B}$. Then, a function

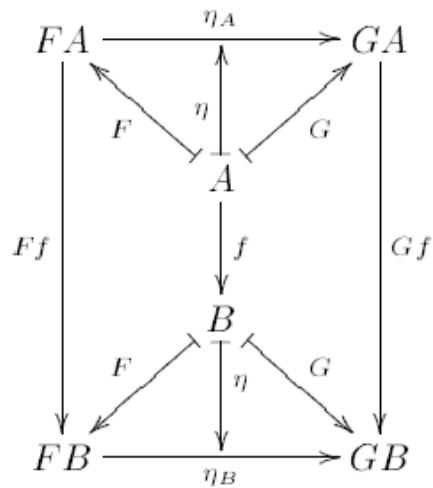
$$\begin{array}{rcl} \eta & : & \mathcal{O}_{\mathcal{A}} \longrightarrow \mathcal{M}_{\mathcal{B}} \\ & & A \mapsto \eta_A \end{array}$$

is called a **transformation** from F to G , iff

$$\forall A \in \mathcal{O}_{\mathcal{A}} \quad \eta_A : FA \xrightarrow{\quad \mathcal{B} \quad} GA ,$$

and it is called a **natural transformation**, denoted $\eta : F \rightarrow G$, iff

$$\forall f : A \xrightarrow{\quad \mathcal{A} \quad} B \quad \eta_B \circ_{\mathcal{B}} Ff = Gf \circ_{\mathcal{B}} \eta_A .$$



The definition says, that a natural transformation *transforms* from a structure F to a structure G , without altering the behaviour of morphisms on objects. I.e., it does not play a role whether a morphism f is lifted into the one or the other structure, when composed with the transformation.
In the drawing on the left, this means that the outer square *commutes*, i.e., all directed paths with common source and target are equal.

Natural transformations in Haskell

First note, that a unary function polymorphic in the same type on source and target side, in fact is a transformation. For example, Haskell's `Just` is typed

```
Just :: forall a. a -> Maybe a .
```

If we imagine this to be a mapping $\text{Just} : \mathcal{O}_{\mathcal{H}} \rightarrow \mathcal{M}_{\mathcal{H}}$, the application on an object $A \in \mathcal{O}_{\mathcal{H}}$ means binding the type variable a to some Haskell type A . That is, `Just` maps an object A to a morphism of type $A \rightarrow \text{Maybe } A$.

To spot a transformation here, we still miss a functor on the source side of `Just`'s type. This is overcome by adding the identity functor $I_{\mathcal{H}}$, which leads to the transformation

```
Just : I_{\mathcal{H}} \rightarrow \text{Maybe} .
```

5.2.1 Lemma $\text{Just} : I_{\mathcal{H}} \rightarrow \text{Maybe}$.

5.2.2 Proof Let $f : A \rightarrow B$ be an arbitrary Haskell function. Then we have to prove, that

$$\text{Just} . I_{\mathcal{H}} f == \text{fmap } f . \text{Just}$$

where the left `Just` refers to η_B , and the right one refers to η_A . In that line, we recognise the definition of the `fmap` for `Maybe` (just drop the $I_{\mathcal{H}}$). \square

Another example, this time employing two non-trivial functors, are the `maybeToList` and `listToMaybe` functions. Their definitions read

```
maybeToList :: forall a. Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just x) = [x]
```

```
listToMaybe :: forall a. [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (x:xs) = Just x .
```

Note, that the *loss of information* imposed by applying `listToMaybe` on a list with more than one element does not contradict naturality, i.e., *forgetting* data is not *altering* data.

Composing transformations and functors

5.3.1 Definition Let $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ be categories, $E : \mathcal{A} \rightarrow \mathcal{B}$, $F, G : \mathcal{B} \rightarrow \mathcal{C}$, and $H : \mathcal{C} \rightarrow \mathcal{D}$. Then, for a natural transformation $\eta : F \rightarrow G$, we define the transformations ηE and $H\eta$ with

$$(\eta E)A := \eta_{EA} \quad \text{and} \quad (H\eta)B := H(\eta_B) ,$$

where A varies over $\mathcal{O}_{\mathcal{A}}$, and B varies over $\mathcal{O}_{\mathcal{B}}$.

Again, due to the above definition, we can write ηEA and $H\eta B$ (for A and B objects in the respective category) without ambiguity. The following pictures show the situation:

$$\begin{array}{ccc} \mathcal{O}_{\mathcal{A}} & \xrightarrow{E} & \mathcal{O}_{\mathcal{B}} \\ & \searrow \eta_E & \downarrow \eta \\ & \mathcal{M}_{\mathcal{C}} & \end{array} \qquad \begin{array}{ccc} \mathcal{O}_{\mathcal{B}} & & \\ \downarrow \eta & \searrow H\eta & \\ \mathcal{M}_{\mathcal{C}} & \xrightarrow{H} & \mathcal{M}_{\mathcal{D}} \end{array}$$

5.3.2 Lemma In the situation of Definition 5.3.1,

$$H\eta : HF \rightarrow HG \quad \text{and} \quad \eta E : FE \rightarrow GE$$

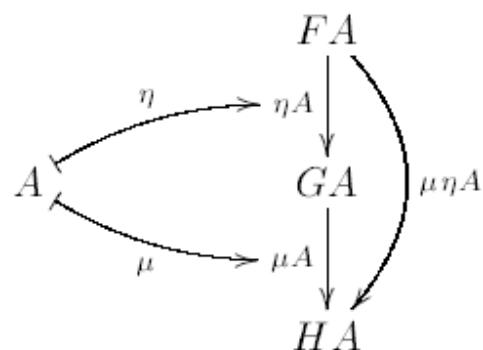
hold. In (other) words, $H\eta$ and ηE are both natural transformations.

Composing transformations and functors

Composing natural transformations Even natural transformations can be composed with each other. Since, however, transformations map objects to morphisms —instead of, e.g., objects to objects— they can not be simply applied one after another. Instead, composition is defined *component wise*, also called **vertical composition**.

5.3.4 Definition Let \mathcal{A}, \mathcal{B} be categories, $F, G, H : \mathcal{A} \rightarrow \mathcal{B}$ and $\eta : F \rightarrow G$, $\mu : G \rightarrow H$. Then we define the transformations

$$\begin{array}{rcl} \text{id}_F & : & \mathcal{O}_{\mathcal{A}} \longrightarrow \mathcal{M}_{\mathcal{B}} \\ & & A \longmapsto \text{id}_{FA} \end{array} \quad \text{and} \quad \begin{array}{rcl} \mu\eta & : & \mathcal{O}_{\mathcal{A}} \longrightarrow \mathcal{M}_{\mathcal{B}} \\ & & A \longmapsto \mu A \circ \eta A . \end{array}$$



The picture on the left (with $A \in \mathcal{O}_{\mathcal{A}}$) clarifies why this composition is called *vertical*.

All compositions defined before Definition 5.3.4 are called **horizontal**. Why?

Monads in theory

6.1.1 Definition Let \mathcal{C} be a category, and $F : \mathcal{C} \rightarrow \mathcal{C}$. Consider two natural transformations $\eta : I_{\mathcal{C}} \rightarrow F$ and $\mu : F^2 \rightarrow F$.

The triple (F, η, μ) is called a **monad**, iff

$$\mu(F\mu) = \mu(\mu F) \quad \text{and} \quad \mu(F\eta) = \text{id}_F = \mu(\eta F) .$$

(Mind, that the parenthesis group composition of natural transformations and functors. They do not refer to function application. This is obvious due to the types of the expressions in question.)

The transformations η and μ are somewhat contrary: While η adds one level of structure (i.e., functor application), μ removes one. Note, however, that η transforms to F , while μ transforms from F^2 . This is due to the fact that claiming the existence of a transformation from a functor F to the identity $I_{\mathcal{C}}$ would be too restrictive:

Consider the set category \mathcal{S} and the list endofunctor L . Any transformation ϵ from L to $I_{\mathcal{S}}$ has to map every object A to a morphism ϵ_A , which in turn maps the empty list to some element in A , i.e.,

$$\begin{aligned} & \epsilon : L \rightarrow I_{\mathcal{S}} \\ \Rightarrow & \forall A \in \mathcal{O}_{\mathcal{S}} \quad \epsilon_A : LA \rightarrow A \\ \Rightarrow & \forall A \in \mathcal{O}_{\mathcal{S}} \quad \exists c \in A \quad \epsilon_A[] = c , \end{aligned}$$

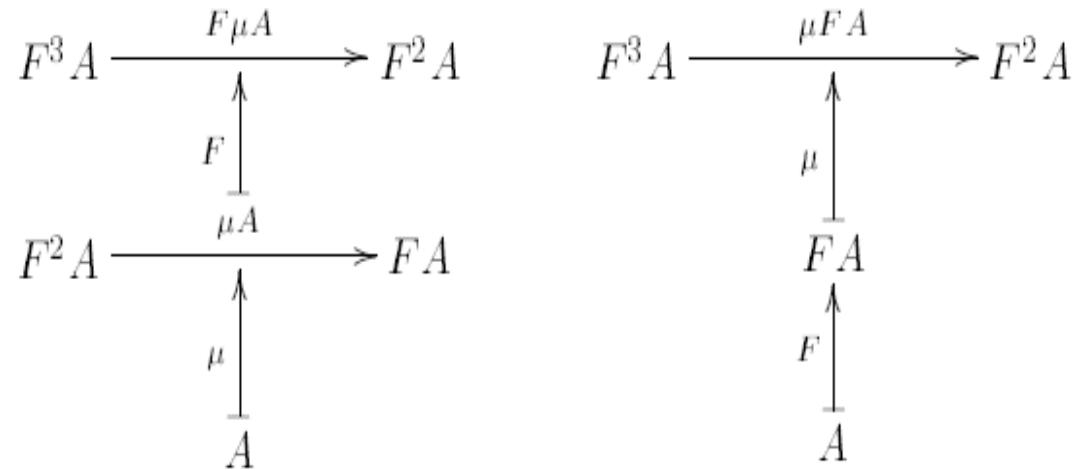
where $[]$ denotes the empty list. This, however, implies

$$\forall A \in \mathcal{O}_{\mathcal{S}} \quad A \neq \emptyset .$$

The following drawings are intended to clarify Definition 6.1.1: The first equation of the definition is equivalent to

$$\forall A \in \mathcal{O} \quad \mu A \circ F\mu A = \mu A \circ \mu FA .$$

So what are $F\mu A$ and μFA ? You can find them at the top of these drawings:



Since application of μA unnests a nested structure by one level, $F\mu A$ pushes application of this unnesting one level into the nested structure. We need at least two levels of nesting to apply μA .

So we need at least three levels of nesting to push the application of μA one level in. This justifies the type of $F\mu A$.

The morphism μFA , however, applies the reduction on the outer level. The FA only assures that there is one more level on the inside which, however, is not touched.

Hence, $F\mu A$ and μFA depict the two possibilities to flatten a three-level nested structure into a two-level nested structure through application of a mapping that flattens a two-level nested structure into an one-level nested structure.

The statement $\mu A \circ F\mu A = \mu A \circ \mu FA$ says, that after another step of unnesting (i.e., μA), it is irrelevant which of the two inner structure levels has been removed by prior unnesting (i.e., $F\mu A$ or μFA).

Let us have a look at the second equation as well. It is equivalent to

$$\forall A \in \mathcal{O}_C \quad \mu A \circ F\eta A = \text{id}_{FA} = \mu A \circ \eta FA .$$

Again, we examine $F\eta A$ and ηFA , which are at the top of the drawings.

$$\begin{array}{ccc}
 FA & \xrightarrow{F\eta A} & F^2 A \\
 \uparrow F & & \uparrow \eta FA \\
 A & \xrightarrow{\eta A} & FA \\
 \uparrow \eta & & \uparrow F \\
 A & & F A
 \end{array}$$

MONADE

- Conceptele din teoria categoriilor traduse în Haskell
- Clasa Functor
- Clasa Monad
- Legi ale monadei
- Modulul Monad
- Studiu de caz

Categoria Hask

- Obiectele sunt tipurile din Haskell
- Morfismele sunt funcțiile din Haskell văzute ca funcții de o singură variabilă
- Constructorii de tip sunt aplicații ce transformă un tip în alt tip
- Funcții de ordin înalt: transformă o funcție în altă funcție

Functori în Hask

- Functorii în Haskell sunt de la aplicații de la **Hask** la *func*, unde *func* este subcategoria lui **Hask** definită pe tipurile acestui functor
 - Functorul listă este un functor de la **Hask** la **Lst**

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

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

Maybe este functor

- Transformă orice tip T într-un nou tip, $\text{Maybe } T$
- Restricția lui fmap la tipurile Maybe transformă o funcție $a \rightarrow b$ într-o funcție $\text{Maybe } a \rightarrow \text{Maybe } b$.
 - $\text{fmap id} = \text{id}$
 - $\text{fmap (f . g)} = \text{fmap f . fmap g}$

Monade

- O *monadă* este un tip special de functor care suportă o structură adițională. Orice monadă este un functor de la o categorie la ea însăși
- O monadă este un functor $M : C \rightarrow C$, împreună cu 2 morfisme atașate fiecărui obiect X din C :
 - $unit^M_X : X \rightarrow M(X)$
 - $join^M_X : M(M(X)) \rightarrow M(X)$

```
class Functor m => Monad m where
  return :: a -> m a
  (=>) :: m a -> (a -> m b) -> m b
```

Monade

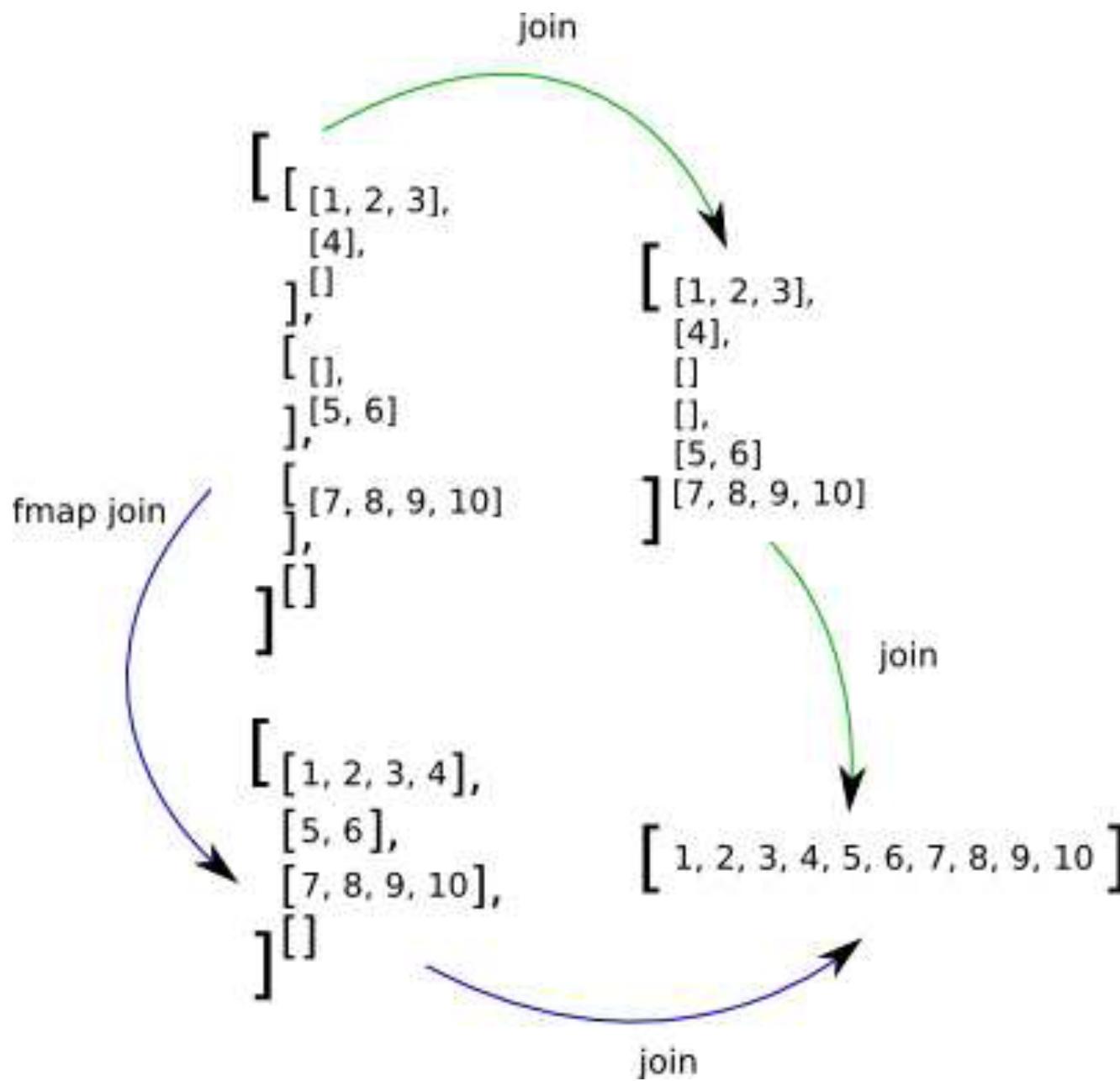
- Se poate defini join și ($>>=$) unul din altul:

```
join :: Monad m => m (m a) -> m a
join x = x >>= id
(>>=) :: Monad m => m a -> (a -> m b) -> m b
x >>= f = join (fmap f x)
```

- Exemplu: functorul powerset $P : \text{Set} \rightarrow \text{Set}$ este o monadă.
Pentru orice multime S :
 - $\text{unit}_S(x) = \{x\}$,
 - $\text{join}_S(L) = \bigcup L$
- P seamănă cu monada listă

Legi ale monadei

- Dată o monadă $M : C \rightarrow C$ și un morfism $f : A \rightarrow B$ unde A, B sunt obiecte în C
 - $\text{join} \circ M(\text{join}) = \text{join} \circ \text{join}$
 - $\text{join} \circ M(\text{unit}) = \text{join} \circ \text{unit} = \text{id}$
 - $\text{unit} \circ f = M(f) \circ \text{unit}$
 - $\text{join} \circ M(M(f)) = M(f) \circ \text{join}$
- În Haskell:
 1. `join . fmap . join = join . join`
 2. `join . fmap return = join . return = id`
 3. `return . f = fmap f . return`
 4. `join . fmap (fmap f) = fmap f . join`



Clasa Monad în Haskell

```
Prelude> :i Monad
class Monad m where
    (">>>=) :: m a -> (a -> m b) -> m b
    (">>>)  :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a
                -- Defined in GHC.Base
instance Monad Maybe -- Defined in Data.Maybe
instance Monad IO -- Defined in GHC.IOBase
instance Monad [] -- Defined in GHC.Base
```

```
Prelude> :i Functor
class Functor f where fmap :: (a -> b) -> f a -> f b
    -- Defined in GHC.Base
instance Functor Maybe -- Defined in Data.Maybe
instance Functor IO -- Defined in GHC.IOBase
instance Functor [] -- Defined in GHC.Base
```

```
Prelude> :i []
data [] a = [] | a : [a]      -- Defined in GHC.Types
instance (Eq a) => Eq [a] -- Defined in GHC.Base
instance Monad [] -- Defined in GHC.Base
instance Functor [] -- Defined in GHC.Base
instance (Ord a) => Ord [a] -- Defined in GHC.Base
instance (Read a) => Read [a] -- Defined in GHC.Read
instance (Show a) => Show [a] -- Defined in GHC.Show
```

Legi ale monadei

- Operațiile unei monade trebuie să îndeplinească trei legi fundamentale:

- **return** este element neutru la stânga pentru $\gg=$

$$(\text{return } x) \gg= f == f \ x$$

- **return** este element neutru la dreapta:

$$m \gg= \text{return} == m$$

- operația $\gg=$ este asociativă:

$$(m \gg= f) \gg= g == m \gg= (\lambda x \rightarrow f \ x \gg= g)$$

- Orice constructor de tip împreună cu **return** și $\gg=$ ce îndeplinește aceste legi este o monadă; programatorul are sarcina să asigure acest lucru la definirea unei noi monade
- Clasa Monad mai are definite două funcții: **fail** și **>>**

Variante monadă a unor funcții pe liste

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
foldM f a [] = return a
foldM f a (x:xs) = f a x >>= \y -> foldM f y xs
```

Studiu de caz: evaluator monadic pentru împărțire

- Expresii cu operatorul de împărțire:

```
data Term = Con Int | Div Term Term  
           deriving Show
```

```
e1, e2 :: Term
```

```
e1 = Div(Div(Con 1972) (Con 2)) (Con 23)
```

```
e2 = Div(Con 2) (Div(Con 1) (Con 0))
```

Evaluator monadic pentru împărțire

- Evaluatorul eval: dacă m este o monadă atunci eval primește la intrare un term și realizează un calcul ce conduce la un întreg astfel:
 - evaluarea lui Con x înseamnă return x
 - evaluarea lui Div t u înseamnă:
 - evaluarea lui t și legarea lui x cu valoarea lui t
 - evaluarea lui u și legarea lui y cu valoarea lui u
 - return x `div` y

```
eval :: Monad m => Term -> m Int
eval(Con x) = return x
eval(Div t u) = do x <- eval t
                    y <- eval u
                    return (x `div` y)
```

Evaluare fără semnalare excepții

- Evaluatorul evalId: specializarea lui eval pentru $m = \text{Id}$
- Monada Id este declarată pentru tipul Id a care este izomorf cu a:
 - return este izomorf cu funcția identitate
 - $\gg=$ este izomorf cu funcția aplicație

```
newtype Id a = MkId a
instance Monad Id where
    return x = MkId x
    (MkId x) >>= q = q x
```

- Pentru a specifica modul de afişare:
`instance Show a => Show(Id a) where
 show(MkId x) = "valoare exp: " ++ show x`
- Evaluatorul de bază:

```
evalId :: Term -> Id Int  
evalId = eval
```

- Exemple:

```
Main> evalId e1  
valoare exp: 42  
Main> evalId e2  
valoare exp:  
Program error: divide by zero
```

Evaluare cu semnalarea excepțiilor

- Tipul Exc a al excepțiilor pentru tipul a:

```
data Exc a = Raise Exception | Return a
type Exception = String
```

- Monada Exc:

```
instance Monad Exc where
    return x = Return x
    (Raise e) >>= q = Raise e
    (Return x) >>= q = q x
    raise :: Exception -> Exc a
    raise e = Raise e
```

Evaluare cu semnalarea excepțiilor

```
evalEx :: Term -> Exc Int
evalEx(Con x) = return x
evalEx(Div t u) = do  x <- evalEx t
                      y <- evalEx u
                      if y == 0
                        then raise "impartire prin zero\n"
                      else return (x `div` y)

instance Show a => Show(Exc a) where
show(Raise e) = "exceptie: " ++ e
show(Return x) = "valoare exp: " ++ show x

Main> evalEx e1
valoare exp: 42
Main> evalEx e2
exceptie: impartire prin zero
```