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 constitue 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, sprezece, zeci :: [String]

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

sprezece =
   ["zece","unsprezece","douasprezece","treisprezece",
   "patrusprezece","cincisprezece","sasesprezece",
   "saptesprezece","optsprezece","nouasprezece"]

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) = sprezece !! 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) şi cea a zecilor(2 cifre) după care se alege şirul corespunzător folosind şi **`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) =”o suta"   -- unu -> o suta
combine3(1, u+1) =”o suta  " ++ convert2(u+1)  -- unu -> o 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 şase cifre

- Numărul de 6 cifre se descompune în două de câte 3 cifre şi 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
”o 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 excepțiile (0, numere negative)

Introducere în IO

- Operațiile de intrare/ieșire în Haskell sunt construite în așa fel încât să nu apară "side-efects"

- 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
  - Acesta 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 întrerupe 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
```

```
Prelude> main = print ([(n, 2^n) |
  n <- [0..19]])
Prelude> main
[(0,1),(1,2),(2,4),(3,8),(4,16),
  (5,32),(6,64),(7,128),(8,256),
  (9,512),(10,1024),(11,2048),
  (12,4096),(13,8192),(14,16384),
  (15,32768),(16,65536),
  (17,131072),(18,262144),
  (19,524288)]
```

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.

```
Prelude> lines "alabalaportocala”
["alabalaportocala"]
Prelude> lines "ala\nbala\nportocala”
["ala","bala","portocala"]
Prelude> unlines ["unu", "doi","trei"]
"unu\ndoi\ntrei\n”

Prelude> words "Facultatea de Informatica”
["Facultatea","de","Informatica"]
Prelude> unwords ["Facultatea", "de",
  "Informatica"]
"Facultatea de Informatica"
```

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
```

Studiu de caz

- Să se transforme un text astfel ca toate cuvintele să înceapă cu literă mare.
- De exemplu:
  - ala bala portocala …. Ala Bala Portocala
  - Universitatea alexandru ioan cuza iasi
  - Universitatea Alexandru Ioan Cuza Iasi

Modulul Data.Char

```
Prelude> import Data.Char

Prelude Data.Char> :i toUpper
toUpper :: Char -> Char -- Defined in ‘GHC.Unicode’

Prelude Data.Char> let uppercase = map toUpper
Prelude Data.Char> uppercase "ala bala portocala”
"ALA BALA PORTOCALA”

Prelude Data.Char> let lowercase=map toLower
Prelude Data.Char> lowercase "ALA BALA PORTOCALA”
"ala bala portocala"
```

```
Prelude Data.Char> :{
Prelude Data.Char| capitalise :: String -> String
Prelude Data.Char| capitalise x =
Prelude Data.Char|    let capWord [] = []
Prelude Data.Char|        capWord (x:xs) = toUpper x : xs
Prelude Data.Char|    in unwords (map capWord (words x))
Prelude Data.Char| :}

Prelude Data.Char> capitalise "ala bala portocala”
"Ala Bala Portocala”

Prelude Data.Char> capitalise "universitatea alexandru ioan
   cuza iasi”
"Universitatea Alexandru Ioan Cuza Iasi”

Prelude Data.Char> capitalise "Universitatea alexandru ioan
   cuza iasi”
"Universitatea Alexandru Ioan Cuza Iasi"
```

Acțiuni - 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 caracterizează clasa Monad (amănunte 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`

- Afișarea 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
11
Prelude> 66^22
107143685717409157344277676895040501186 56
Prelude> it
107143685717409157344277676895040501186 56
```

it

```
Prelude> import Data.Time.Calendar.Easter
Prelude Data.Time.Calendar.Easter> orthodoxEaster 2016
2016-05-01
Prelude Data.Time.Calendar.Easter> orthodoxEaster 2017
2017-04-16
Prelude Data.Time.Calendar.Easter> it
2017-04-16
```

Definirea de tipuri de date

- Haskell are trei modalități de a introduce tipuri (tipuri utilizator):
  - Declarații **`data`** pentru *enumerări* și *structuri* (abstracte) 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 (==, /=) şi 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 == şi /=)

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]
```

```
Prelude> succ 7
8
Prelude> pred 9
8
Prelude> fromEnum 'b'
98
Prelude> take 5 (enumFrom 2)
[2,3,4,5,6]
Prelude> take 5 (enumFromThen 2 13)
[2,13,24,35,46]
Prelude> take 5 (enumFromTo 2 13)
[2,3,4,5,6]
Prelude> enumFromThenTo 2 4 22
[2,4,6,8,10,12,14,16,18,20,22]
```

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 şi 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 parametrizate

```
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 parametrizate

```
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 inregistrarea 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 u_1 … u_k = K_1 t_{11}…t_{1k1} | … | K_n t_{n1}…t_{nkn}

- introduce un nou constructor de tip T
- unul sau mai mulți constructori de dată constituenți K_1, …, K_n
- t_{ij} – tipuri, cx - context

- Tipul constructorilor de dată (funcții):

  K_i :: t_{i1} -> t_{i2} -> … t_{iki} -> (T u_1 u_2… 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 Distanta = 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 numeşte
- 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 şi 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 data “arbori binari”

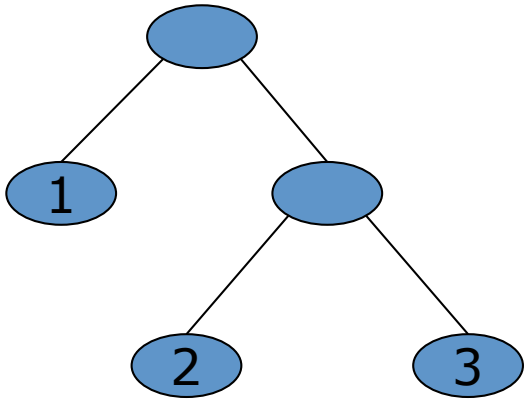
- O valoare a tipului **`Btree a`** este fie:
  - un nod frunză (Leaf) ce conține o valoare de tip **`a`**
  - un nod ramificație (Fork) şi doi noi arbori, subarborele stâng al nodului ramificație respectiv cel drept
  - o frunză se numeşte nod exterior
  - un nod ramificație se numeşte nod interior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

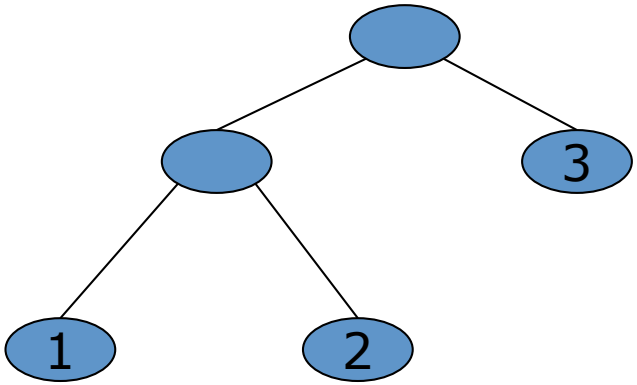
Structura de date arbori binari

- 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(Leaf x)
  - P(xt), P(yt)  P(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 = lenght.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 înlocuieşte 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))
```

Arbori perfecți

- Un arbore binar se zice *perfect* dacă toate frunzele sale au aceeaşi 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)

- **`height xt < size xt <= 2`**$^{\texttt{height xt}}$

- **`⌈log(size xt)⌉ <= height xt < 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 şi se construieşte 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** şi **foldBtree**

- Sunt funcțiile analoge funcțiilor map şi 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`** şi **`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)
```

Exemple

- 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'
```

Exemple

- 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
```