Cursul 4 - Liste (continuare)

- O altă reprezentare a listelor
  - Generatori
  - Gărzi
- zip, unzip
- foldr, foldl
- Studii de caz

Liste: o altă notaţie

- Analogie cu definirea unei mulţimi:

  $\{1, 4, 9, 16, 25\} = \{x^2 \mid x \in \{1..5\}\}$

- List comprehension

```
Hugs.Base> [x*x|x<-[1..5]]
[1,4,9,16,25]
Hugs.Base> [x*x|x<-[1..5], odd x]
[1,9,25]
Hugs.Base> [x*x|x<-[1..10], even x]
[4,16,36,64,100]
```

Liste

- Sintaxa Haskel pentru definirea listelor:

  [e | Q] unde

  - e este o expresie
  - Q este un calificator
    - O secvenţă – posibil vidă – de forma

      gen1, gar1, gen2, gar2, ...
    - 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 | Q] calificatorul 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

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

Hugs> [(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)]

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

Hugs> [(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)]

Hugs> [(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]

Main> quicksort [3,2,5,4,2,6,4,5]
[2,2,3,4,4,5,5,6]
Main> quicksort ["luni", "marti","miercuri", "joi",
   "vineri"]
["joi","luni","marti","miercuri","vineri"]
```

Proprietăţi

```
[f x|x <- xs]      = map f xs

[x|x <- xs, p x]   = filter p xs

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

[e|Q,x<-[d|P]]     = [e[x:=d]|Q,P]
```

Funcţia zip

- Listele xs şi ys sunt transformate într-o listă de perechi cu elemente de pe acelaşi 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')]
```

Aplicaţii ale funcţei `zip`

- Produsul scalar:

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

- Căutare (poziţiile lui x în lista xs):

```
positions :: (Eq a) => a -> [a] -> [Int]
position x xs = [i|(i,y)<-zip[0..]xs, x == y]

Main> positions 3 [1,2,3,4,3,2,5,3,3,5,4,3]
[2,4,7,8,11]
```

Aplicaţii ale funcţei `zip`

- Funcţia **`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)]
```

- Funcţia **`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)]
```

Aplicaţii ale funcţei `zip`

- este o secvenţă 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
```

Funcţia **unzip**

- O listă de perechi [(x1,y1),(x2, y2),…]
  este transformată într-o pereche de 2 liste ([x1,x2,…], [y1, y2,…])

```
pair:: (a -> b, a -> b) -> a -> (b, b)
pair (f, g) x = (f x, g x)

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

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

Funcţia cross

- Denumirea pentru f×g din teoria categoriilor

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

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

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

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

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

Funcţia cross

- Proprietăţi
  1. fst.pair(f,g) = f
  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)

Demonstraţie 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)
```

Funcţile zip, unzip, cross

- Proprietăţi:

```
cross(map f, map g).unzip = unzip.map(cross(f, g))
```

```
Test> (cross(map (+2), map (*3)).unzip) [(1,2), (3,4)]
([3,5],[6,12])
Test> (unzip.map(cross((+2), (*3)))) [(1,2), (3,4)]
([3,5],[6,12])
```

Funcţia `foldr` (fold right)

- Utilizată pentru simplificarea definiţiilor recursive de forma (@ este o operaţie):

```
h []          = e
h (x:xs)      = x @ h xs
```

- Această funcţie transformă lista x1:(x2:(x3:(x4:[]))) în x1@(x2@(x3@(x4@e)))

- Şablonul din definiţia lui h este capturat în funcţia:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

- Astfel `h = foldr (@) e`
- `foldr(@)e[`x_0`,`x_1`,…`x_n`]=`x_0`@(`x_1`@(...(`x_n`@e)...))`

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
```

Funcţia `foldl` (fold left)

- Utilizată pentru simplificarea definiţiilor recursive de forma (@ este o operaţie):

```
h e []       = e
h e (x:xs)   = h(e@x)xs
```

- Această funcţie transformă lista x1:(x2:(x3:(x4:[]))) în (((e@x1)@x2)@x3)@x4
- Şablonul din definiţia lui h este capturat în funcţia:

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

- Astfel `h = foldl (@)`
- `foldl(@)e[`x_0`,`x_1`,…`x_n`]=(…((e@`x_0`)@`x_1`)..`x_{n-1}`)@`x_n

Exemple

```
sum = foldr(+)0

sum =     sum1 0
          where
                sum1 e [] = e
                sum1 e (x:xs)  = sum1(e+x) xs
sum = foldl(+) 0
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
```

Studiu de caz 1: `maxlist`

- Elementul maxim dintr-o listă de elemente ce aparţin 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
```

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 "aseewweeree"
'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`

- Soluţia 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(⊕)[x0,x1,x2,x3] = x0⊕(x1⊕(x2 ⊕x3))
foldl1(⊕)[x0,x1,x2,x3] = ((x0⊕x1)⊕x2)⊕x3

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

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

Studiu de caz 2: Conversii

Proiectaţi o funcţie **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 pronunţia lui n în limba română (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"

Cazul numerelor cu două cifre

- 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 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 cifra) 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)
combine3(1, 0) ="una suta"
combine3(1, u+1) ="una suta  " ++ convert2(u+1)
combine3(2, 0) ="doua sute"
combine3(2, u+1) ="doua sute  " ++ convert2(u+1)
combine3(t+2, 0) = unitati!!(t+1) ++ " 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)
combine6(1, 0) ="una suta mii"
combine6(1, u+1) ="una suta mii si " ++ convert3(u+1)
combine6(2, 0) ="doua sute mii"
combine6(2, u+1) ="doua sute 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  sapt
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"
```