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)

λ 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):

$$Exp ::= Var \mid Exp\ Exp \mid \lambda Var.Exp \mid (E)$$

unde Var are valori din V

- Exemple:

$$\lambda x.x,\ \lambda x.xx,\ \lambda x.(fx)(gx),\ (\lambda x.fx)x$$

- Notații:
  - $FM_1M_2\ldots M_n$ notează $(\ldots((FM_1)M_2)\ldots M_n)$
  - $\lambda x_1x_2\ldots x_n.M$ notează $\lambda x_1.(\lambda x_2.(\ldots\ (\lambda x_n.\ (M))\ldots))$

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

- Mulțimea 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:
  - $y[x := E'] =$ if $y = x$ then E' else y
  - $(E1E2)[x := E'] = (E1[x := E'])(E2[x := E'])$
  - $(\lambda x.E)[x := E'] = \lambda x.E$
  - $(\lambda y.E)[x := E'] =$

$$= \lambda y.(E[x := E']) \qquad \text{if } y \notin FV(E')$$

$$= \lambda z.((E[y := z])[x := E']) \qquad \text{if } y \in FV(E')$$

Lambda calcul - formal

- M este term închis, sau combinator, dacă $FV(M) = \Phi$. Mulțimea 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 in 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?)

$(yxzx(\lambda x.yx)x)[x:=abc] \rightarrow$

$y(abc)z(abc)(\lambda x.yx)(abc)$

Exemplul 2

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

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

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

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

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

cc

Exemplul 3

$(\lambda xa.xa)(\lambda x.xa) \to \lambda a.(\lambda x.xa)a \to \lambda a.aa$

In $\lambda a.(\lambda x.xa)a$, a a devenit legat si era liber! $(\lambda x.xa)$

Corect: (se aplică o reducere alfa)

$\lambda xa.xa = \lambda xb.xb$

$(\lambda xb.xb)(\lambda x.xa) \to \lambda b.(\lambda x.xa)b \to \lambda b.ba$

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:

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

$(\lambda x.xxx)(\lambda x.xxx) \rightarrow (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \rightarrow \ldots$

Forme normale

- Pot exista mai multe strategii de reducere:

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

$(\lambda gx.\ \underline{(\lambda a.a)(gx)}\ )(\lambda b.bb)c \quad \rightarrow ?$

$(\lambda\ f.(\lambda\ x.fx)y)g \ \rightarrow (\lambda\ f.fy)g$

$(\lambda\ f.(\lambda\ x.fx)y)g \ \rightarrow (\lambda\ x.gx)y$

Forme normale

- **Teorema confluenței**: Relația → este confluentă: dacă E →* E1 și E →* E2 atunci există E' astfel ca E1 →* E' și E2 →* 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 $\forall F \in \Lambda$ $F(YF) = YF$

<u>Demonstrație</u>

(1) Fie $W \equiv \lambda x.F(xx)$ si $X \equiv WW$ Atunci

$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv F(X)$

(2) La fel

Exemplu

- Fie:

  F = λ f. λ x.(if x == 0 then 1 else x * f(x - 1))

  (Y F) 3 =
  F (Y F) 3 =
  (λ f. λ x.(if x == 0 then 1 else x * f(x - 1))) (Y F) 3)
  if 3 == 0 then 1 else 3 * (Y F)(3 - 1))
  3 * ((Y F) 2)
  ...
  6 * ((Y F) 0)
  6 * (F (Y F) 0) =
  6 * ((λ f. λ x.(if x == 0 then 1 else x * f(x - 1))) (Y F) 0))
  6 * if 0 == 0 then 1 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: $c_0, c_1, \ldots\ \ c_n \equiv \lambda fx.\ f^n x$

Operații: $A_+ \equiv \lambda xypq.xp(ypq)$

$A_* \equiv \lambda xyz.x(yz)$

$A_{exp} \equiv \lambda xy.yx$

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

Reprezentarea funcțiilor

<u>Definiție</u>.

`true` $\equiv K \equiv \lambda xy.x$, `false` $\equiv K_* \equiv \lambda xy.y$

`if-then-else-` $\equiv \lambda xyz.xyz$

`if` B `then` P `else` Q poate fi reprezentat prin BPQ

and $\equiv \lambda$ xy.(x y `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 $\underline{0}$ = true, Zero $\underline{n+1}$ = false

Demonstratie:

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

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

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

Reprezentarea funcțiilor

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

$F\ \underline{n}_1\ \underline{n}_2 \ldots \underline{n}_p = \underline{f(n_1, n_2, \ldots, n_p)}$

<u>Teoremă</u> Funcțiile recursive sunt λ-definibile:

(1) Funcțiile inițiale(zero, succesor, proiecția) sunt λ-definibile

(2) Clasa funcțiilor λ-definibile este închisă la compunere($f(n) = g(h_1(n),\ldots,h_m(n))$)

(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
```

Exemple

```
Prelude> let f = \x -> x*x
Prelude> f 3
9
Prelude> let g = \z ->z.z
Prelude> g f 3
81
```

Exemple

- 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*2 + 1
```

- Alternativă (lambda expresie):

```
odds n = map(\x -> x*2 + 1)[0..n-1]
```

Secțiuni

- Funcțiile cu două argumente pot fi utilizate infix(ca operatori): **`12 `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 şi sunt funcții:

```
(@) = \x ->(\y -> x @ y)
(x @) = \y -> x @ y
(@ y) = \x -> 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+) = \y -> 1+y
(1/) = \y -> 1/y
(*2) = \x -> x*2
(/2) = \x -> x/2
```

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

Definiții recursive

- O funcție care are în expresia ce defineşte 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 3.9999999999999999
24.0
```

Definiții recursive

- O funcție recursivă se defineşte astfel:
  - Se defineşte 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 “patern 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 patern matching
  - Folosește gărzi în definirea funcțiilor
- Stilul expresie
  - Folosește expresii let
  - Folosește lambda abstracții
  - Folosește expresii case
  - Folosește expresii if

Liste

- Tipul listă

```
Prelude> :t [1,2,3]
[1,2,3] :: Num t => [t]
Prelude> :t [[1,2,3,4],[1,2]]
[[1,2,3,4],[1,2]] :: Num t => [[t]]
Prelude> :t [(+),(*),(-)]
[(+),(*),(-)] :: Num a => [a -> a -> a]
Prelude> :t [(+),(*),(-),(/)]
[(+),(*),(-),(/)] :: Fractional a => [a -> a -> a]
```

- Constructorul (cons) :

```
Prelude> :t (:)
(:) :: a -> [a] -> [a]

Prelude> 1:[2,3,4]
[1,2,3,4]
Prelude> 1:2:3:[]
[1,2,3]
Prelude> 1:(2:(3:[]))
[1,2,3]
```

Liste

- Funcția **null**

```
null   :: [a] -> Bool
null[]       = True
null x:xs    = False
```

```
Prelude> null []
True
Prelude> null [1,2,3]
False
```

Operații cu liste

- Concatenare: ++

```
(++)      :: [a] -> [a] -> [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 ++ ys) ++ zs = xs ++ ( ys ++ zs)
xs ++ [] = [] ++ xs
```

- Funcția `concat` concatenează o listă de liste:

```
    concat :: [[a]] -> [a]
    concat [] = []
    concat(xs:xss) = xs ++ 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) paşi

```
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”
Prelude> reverse "IONALANICAIIACINALANOI”
"IONALANICAIIACINALANOI"
```

Funcția **length**

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

- Se poate dovedi (de la definițiile ++ şi **length**):
  length(xs++ys) = length xs + length ys
- Funcția **length** calculează lungimea unei liste în n paşi 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

- Compunere de funcții (.) :

```
(.) :: (a -> b) -> (c -> a) -> c -> b

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

Funcțiile `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]
```

Funcția **`take`**

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

```
take     :: Int -> [a] -> [a]
```

  - Enumerarea cazurilor: sunt 2 valori pentru primul argument: 0 şi n+1, două pentru al doilea argument: [] şi x:xs

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

  - Definirea cazurilor de bază:

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

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
```

Funcțiile **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 :

```
take n xs ++ drop n xs       = xs
take m .take n               = take(m `min` n)
drop m .drop n               = drop(m+n)
take m .drop n               = drop n. take(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 şi 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

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

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 | Q]   unde

  – e este o expresie

  – Q este un calificator

    - O secvență – 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

```
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 <- 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]
```

Funcția `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')]
```

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

```
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×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):

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

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

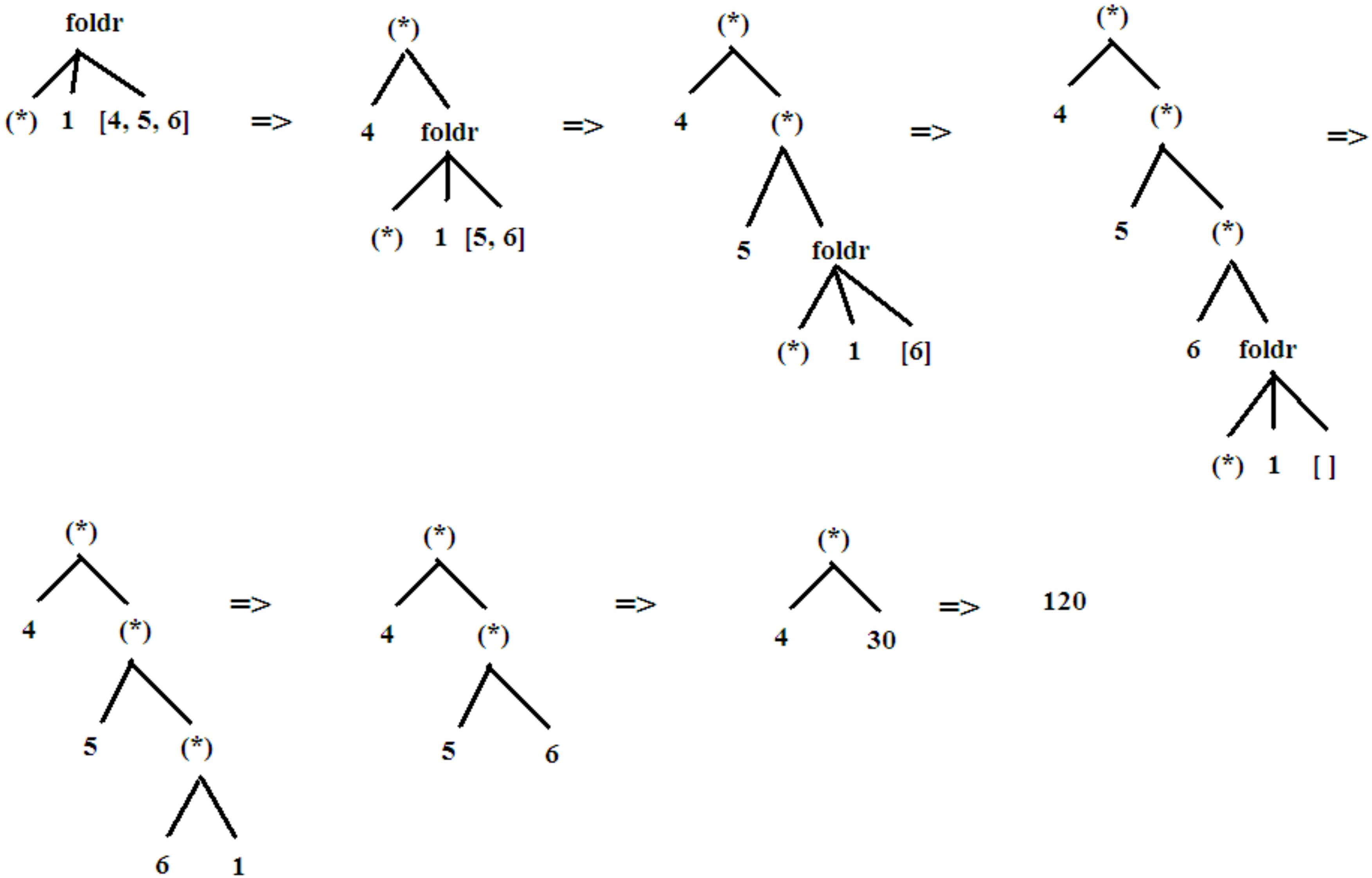
- Şablonul din definiția lui h este capturat în funcția:

```
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)...))`

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

```
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):

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

- Această funcție transformă lista x1:(x2:(x3:(x4:[]))) în (((e@x1)@x2)@x3)@x4
- Şablonul din definiția lui h este capturat în funcția:

```
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 (@) e`
- **foldl (@) e** $[x_0, x_1, \ldots x_n] = (\ldots((e@x_0)@x_1)\ldots x_{n-1})@x_n$

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 =    sum1 0
         where
               sum1 e [] = e
               sum1 e (x:xs) = sum1 (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 -> b -> 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 "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**

- 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(⊕)[x0,x1,x2,x3] = x0⊕(x1⊕(x2 ⊕x3))
foldr1 (/) [1, 2, 3] = 1.5

foldl1(⊕)[x0,x1,x2,x3] = ((x0⊕x1)⊕x2)⊕x3
foldl1 (/) [1, 2, 3] = 0.16666666666667

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

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