

Logică și structuri discrete

## Liste

Marius Minea

marius@cs.upt.ro

<http://www.cs.upt.ro/~marius/curs/lsd/>

6 octombrie 2014

# Liste

Listele sunt un mod de bază de a lucra cu *colecții* de elemente.

O listă e o secvență *finită*, *ordonată*, de valori de *același tip*.

listele sunt *finite*, dar pot avea lungime oricât de mare  
spre deosebire de *tuple* (tip separat pentru fiecare lungime):  
perechi, triplete, *n*-tuple  
nu pot fi infinite (altă noțiune/tip: engl. *stream*)

*ordinea* elementelor contează:  $[1; 3; 2] \neq [3; 1; 2]$   
spre deosebire de *mulțimi*

*accesul* la elementele listei e *secvențial* (acces direct doar la primul)  
spre deosebire de *tablou*, cu *acces direct* la toate elementele

## Listele ca tip recursiv

Listele pot fi *definite recursiv*:

O listă este o listă vidă, sau un *element* urmat de o listă.

Mai precis, pentru că listele sunt de un anumit *tip*:

- lista vidă este o listă de elemente de tip *a* (pentru orice tip)
- combinația dintre un element de tip *a* (ca prim element) și o listă de elemente de tip *a* (restul = coada listei) e deasemenea o listă de elemente de tip *a*.

Definiția de mai sus se numește și *inductivă*, deoarece definește toate listele (de un anumit tip):

- pornind de la cea mai simplă (*cazul de bază*),
- și prin modul de a construi o listă mai mare dintr-una mai mică (*pasul inductiv*).

## Lista ca tip de date în ML

ML definește tipul de date 'a list pentru orice tip de element 'a  
Listele se scriu între paranteze drepte [ ] cu ; între elemente:

```
# [1;3;2] (* o lista de intregi *);;
```

```
- : int list = [1; 3; 2]
```

```
# ["asta";"e";"o";"lista"] (* o lista de siruri *) ;;
```

```
- : string list = ["asta"; "e"; "o"; "lista"]
```

*Operatorul ::* creează o listă dintr-un element (care devine capul noii liste) și o listă (care devine coada noii liste). *Lista vidă* e [].  
:: e asociativ la dreapta (se poate folosi de mai multe ori).

```
# 2 :: [3;1];;
```

```
- : int list = [2; 3; 1]
```

```
# 1 :: 2 :: [];;
```

```
- : int list = [1; 2]
```

@ concatenează două liste. Evitați, trebuie să parcurgă prima listă!

```
# [2;1] @ [3];;
```

```
- : int list = [2; 1; 3]
```

## Potrivirea de tipare (pattern matching)

Un tip definit prin mai multe variante (listă vidă sau `cap+coadă`) se prelucrează prin *tipare* (cu un tipar pentru fiecare variantă).

Potrivirea de tipare ne permite să descompunem un obiect după *structură*, identificând (și numind) părțile componente.

```
match lista with
| [] -> expr1
| cap :: coada -> expr2
```

Această construcție e o *expresie*, cu rezultat `expr1` dacă `lista` e vidă; altfel, identificatorii `cap` și `coada` sunt *legați* la cele două părți ale listei, și pot fi folosiți în `expr2`, care dă rezultatul întregii expresii

Mai concis, cu *function* definim o *funcție de argument listă* cu rezultatul dat de valoarea expresiei–tipar:

```
function
| [] -> expr1
| cap :: coada -> expr2
```

Bara | la prima variantă e opțională dar poate face codul mai ușor de citit

## Funcții predefinite cu liste

Modulul `List` oferă o serie funcții pentru lucrul cu liste.

Le folosim cu numele `List.numefuncție`.

(e preferabil față de deschiderea modului `open List` și folosirea numelui simplu, pentru că e mai clar în cod că se folosesc liste)

Funcțiile cele mai simple: `List.hd` (*head*) și `List.tl` (*tail*)  
returnează capul, și respectiv coada listei.

```
# List.hd [1;4;3];;  
- : int = 1  
# List.tl [1;4;3];;  
- : int list = [4; 3]
```

Alte funcții: `length`, `mem` (membru), `nth` (al n-lea element), etc.

## Potrivire de tipare sau `hd` / `tl` ?

Funcțiile `hd` și `tl` nu sunt definite pentru lista vidă (sunt funcții *parțiale* pe tipul listă). La apel cu `[]` ele *generează o excepție*.

```
# List.hd [];;
```

```
Exception: Failure "hd".
```

```
# List.tl [];;
```

```
Exception: Failure "tl".
```

Vom discuta în alt curs despre tratarea excepțiilor.

Într-o funcție, trebuie să tratăm *toate cazurile*.

Folosind potrivirea de tipare (cu `match ... with` sau `function`) *compilatorul verifică* și ne asigură că nu am uitat nicio variantă.

Folosind `hd` și `tl` trebuie să ne asigurăm noi că nu avem lista vidă.

Preferăm de aceea accesul la cap/coadă prin *potrivirea de tipare*.

## Funcții simple: lungimea unei liste

```
let rec len = function
  | [] -> 0
  | _ :: t -> 1 + len t
```

Tiparul `_` se potrivește cu orice. Folosim pentru a ignora valoarea.

Varianta: în parcurgere, acumulăm rezultatul parțial

⇒ încă un parametru: elementele numărate până acum

```
let rec len2 r = function
  | [] -> r
  | _ :: t -> len2 (r+1) t
```

```
let len = len2 0 (* urmatorul argument e lista *)
```

(inițial, nu am numărat niciun element, de aici argumentul 0)

Modulul `List` definește funcția `List.length`.



## Definiții locale

```
let rec len2 r = function
  | [] -> r
  | _ :: t -> len2 (r+1) t
let len = len2 0
```

se rescrie mai frumos:

```
let len lst =
  let rec len2 r = function
    | [] -> r
    | _ :: t -> len2 (r+1) t
  in len2 0 lst
```

Putem citi în felul următor: `let len lst = len2 0 lst` unde definiția funcției `len2` e dată (și vizibilă) doar în interior.

Doar funcția `len2` e recursivă, nu și `len` (doar folosește `len2`).

## Recursivitatea finală (prin revenire; tail recursion)

```
În varianta      let rec len = function
                  | [] -> 0
                  | _ :: t -> 1 + len t
```

adunarea se face doar la revenirea din apel: pentru o listă de lungime  $n$  vom avea  $n$  apeluri active când ajungem la sfârșitul ei.  
⇒ *va eșua prin depășirea stivei* la liste foarte lungi

```
let len lst =
  let rec len2 r = function
    | [] -> r
    | _ :: t -> len2 (r+1) t
  in len2 0 lst
```

Preferăm varianta 2:

Apelul recursiv e *ultima* operație pe acea ramură (*tail recursion*).  
⇒ fiecare apel recursiv returnează *aceeași valoare* ca cel anterior  
Nu sunt necesare calcule la revenire și stiva, rezultatul poate fi returnat direct. Compilatorul va *optimiza* apelul în *iterație* (ciclu)

*Recursivitatea finală e practic la fel de eficientă ca iterația*

## Funcții simple: testul de membru

Apare valoarea  $x$  în listă ?

```
let rec mem x = function
```

```
  | [] -> false
```

```
  | h :: t -> x = h || mem x t
```

```
val mem : 'a -> 'a list -> bool = <fun>
```

$x$  e membru în listă dacă:

$x$  e capul listei  $h$ , SAU

$x$  e membru în coada listei  $t$

Operatorul `||`: SAU logic: cel puțin un operand adevărat (`true`)  
dacă primul e `true`, rezultatul e `true`  $\Rightarrow$  nu mai evaluează al doilea

`List.mem` e deasemenea o funcție predefinită .

## Putem defini propriul tip listă ?

Limbajele funcționale au teoria tipurilor bine fundamentată matematic; ele permit definirea de noi tipuri după anumite reguli.

Un *tip de date algebric* e definit ca o combinație de alte tipuri.

Variante comune sunt:

Tipul *produs* (produsul cartezian  $A \times B$  a două tipuri  $A$  și  $B$ )

```
type physunit = float * string
```

O valoare a acestui tip e o pereche (în general: tuplu) de elemente din tipurile componente.

Tipul *sumă* (tip uniune, sau tip cu *variante*).

În ML, fiecare variantă are un *constructor* de tip, notat printr-o etichetă (tag), pentru a deosebi variantele între ele

```
type mix = Int of int | Real of float | Str of string
```

## Listele ca tip de date algebric

Pentru a da definiția recursivă a tipului listă, ca mai sus, ne trebuie:

- un *constructor* Cons care creează o nouă listă dintr-un element (capul listei) și altă listă (coada noii liste)
- o valoare (de fapt constructor fără argumente) pentru *lista vidă*

Am putea defini *tipul recursiv* listă de elemente de tip 'a ca fiind:

```
type 'a list = Nil | Cons of 'a * 'a list
```

(un tip sumă dintre un tip cu doar o valoare, și un tip produs)

Putem prelucra o valoare de acest tip prin potrivire de tipare:

```
let rec len lst = match lst with  
  | Nil -> 0  
  | Cons (_, t) -> 1 + len t
```

În ML tipul listă e *predefinit* în acest fel:

constructorul de listă vidă e []

constructorul cu două argumente e scris ca operator infix ::

## Lista ca tip de date abstract (TDA)

Un *tip de date abstract* e un tip de date definit prin operațiile care se pot efectua asupra lui (și constrângerile între acestea), fără a expune modul în care acestea sunt implementate.

TDA *izolează* (abstractizează) *definiția/interfața* de *implementare*.

Ne permite implementări modificabile și interschimbabile, fără a afecta programul care le folosește doar prin interfață.

TDA listă  $L$  cu tip de element  $E$  e de obicei definit prin:

$nil : () \rightarrow L$  (constructorul de listă vidă)

$cons : E \times L \rightarrow L$  (constructorul pentru liste)

$hd : L \rightarrow E$  (capul listei)

$tl : L \rightarrow L$  (coada listei)

și axiomele  $hd(cons(e, l)) = e$  și  $tl(cons(e, l)) = l$

Această definiție *abstractă* e suficientă pentru a defini lista ca obiect matematic și a raționa despre proprietățile listelor.

## Parcurgeri standard pentru liste

E natural să *prelucrăm toate elementele* unei liste.

Distingem trei cazuri principale

1. *Facem* ceva pentru fiecare element (ex. tipărim)  
(fără a produce vreo valoare ca rezultat) `List.iter`
2. *Transformăm* fiecare element al listei cu aceeași funcție  
obținem o nouă listă `List.map`
3. *Combinăm* toate valorile din listă `List.fold_left`  
(le acumulăm succesiv într-un rezultat) `List.fold_right`

Acestea sunt *funcții de iterare* pentru liste (a itera = a repeta).

Scriem doar funcția pentru *un pas* de prelucrare (un element),  
iar lista e *parcursă automat* de funcțiile standard de iterare.

## Iterarea peste toate elementele listei

`List.iter` : ('a -> unit) -> 'a list -> unit

`List.iter f [e1; e2; ... en]` apelează `f e1`; `f e2`; ... `f en`

Funcția `f`: folosită pentru *efectul* (ex. tipărire), nu valoarea ei  
ML are tipul `unit`, cu singura valoare notată `()` (C are `void`)

```
let rec iter f = function
  | [] -> () (* orice functie are valoare, aici () *)
  | h :: t -> f h; iter f t (* a; b are valoarea b *)
```

sau, cu o funcție auxiliară care evită retransmiterea parametrului `f`:

```
let iter f =
  let rec iter1 = function
    | [] -> ()
    | h :: t -> f h; iter1 t
  in iter1
```

`List.iter print_int [1;2;3]` (\* tipareste 123 fara spatii \*)

`List.iter (Printf.printf "%d ") [1;2;3]`

`1 2 3 - : unit = ()` (\* tipareste 1 2 3 cu spatii \*)



## Transformarea tuturor elementelor dintr-o listă

`List.map` : (`'a` -> `'b`) -> `'a list` -> `'b list`

`List.map` [`e1`; `e2`; ... `en`] e lista [`f e1`; `f e2`; ... `f en`]

Rezultatul lui `f` poate avea alt tip decât parametrul  
putem obține o listă cu alt tip de elemente

```
let rec map f = function
```

```
  | [] -> []
```

```
  | h :: t -> f h :: map f t
```

sau, cu o funcție auxiliară

```
let map f =
```

```
  let rec map1 = function
```

```
    | [] -> []
```

```
    | h :: t -> f h :: map1 t
```

```
  in map1
```

[ `e1` ; `e2` ; ... `en` ]



[ `r1` ; `r2` ; ... `rn` ]



...



```
List.map ((+) 2) [3; 7; 4]
```

```
(* lista [5; 9; 6] *)
```

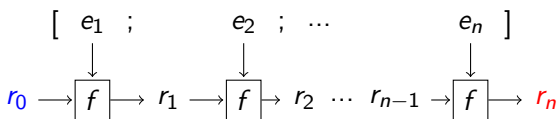
```
List.map String.length ["acesta"; "e"; "un"; "test"]
```

```
- : int list = [6; 1; 2; 4] (* lista lungimilor sirurilor *)
```

## Combinarea elementelor dintr-o listă (de la cap)

`List.fold_left` : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

`List.fold_left` f r0 [e1; e2; ... en] = f (...f (f r0 e1) e2...)



```
let fold_left f =
  let rec fold1 a = function
    | [] -> a
    | h :: t -> fold1 (f a h) t
  in fold1
```

`List.fold_left` prelucrează elementele de la cap, e *tail-recursive*.

`List.fold_left` (+) 0 [3; 7; 4] (\* suma pornind de la 0: 14 \*)

`List.fold_left` (fun s e -> s + String.length e) 0 ["a"; "si"; "b"]  
- : int = 4 (\* suma lungimilor sirurilor din lista \*)

0 + length "a" = 1 → 1 + length "si" = 3 → 3 + length "b" = 4

## Cum funcționează și cum folosim `List.fold_left`

`List.fold_left`: un *ciclu*, odată pentru fiecare element al listei calculează un *rezultat, actualizat la fiecare iterație* (pentru fiecare element)

`List.fold_left` are nevoie de 3 parametri:

- o funcție `f` cu 2 parametri de tip `'a -> 'b -> 'a`  
p1: rezultatul calculat până acum de tip `'a`  
p2: elementul curent din listă de tip `'b`  
rezultatul `f(p1, p2)` devine p1 în apelul cu următorul element
- valoarea inițială de tip `'a`  
(rezultatul pentru lista vidă, și p1 pentru primul apel al lui `f`)
- lista de prelucrat de tip `'b list`

Un limbaj imperativ ar folosi o *variabilă, atribuită* la fiecare iterație  
În `List.fold_left`, rezultatul funcției `f` în fiecare iterație e folosit de `f` în următoarea iterație (ca prim parametru).

## Exemple de funcționare pentru List.fold\_left

Minimul unei liste

```
let list_min = function
  | [] -> invalid_arg "empty list" (* exceptie *)
  | h :: t -> List.fold_left min h t

list_min [3; 9; -2; 4]    -> List.fold_left min 3 [9; -2; 4]
(fun m e -> min m e) 3 9    -> min 3 9 = 3
(fun m e -> min m e) 3 (-2) -> min 3 (-2) = -2
(fun m e -> min m e) (-2) 4 -> min (-2) 4 = -2
```

Inversarea unei liste:

capul listei rămas de inversat devine capul rezultatului acumulat

```
let rev = List.fold_left (fun t h -> h :: t) [] [3; 7; 5]
(fun t h -> h :: t) [] 3    -> 3 :: [] = [3]
(fun t h -> h :: t) [3] 7   -> 7 :: [3] = [7; 3]
(fun t h -> h :: t) [7; 3] 5 -> 5 :: [7; 3] = [5; 7; 3]
```

List.rev există ca funcție standard pentru liste

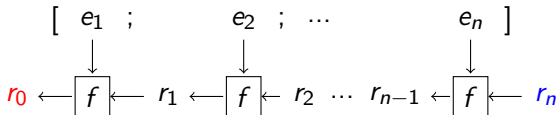
## Combinarea elementelor dintr-o listă (de la coadă)

`List.fold_right` : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

`fold_right f [e1; e2;...; en] rn = f e1 (f e2 (...(f en rn)...))`

`fold_right` calculează, de la dreapta la stânga rezultatul

`r0 = f e1 r1` ← `r1 = f e2 r2` ← ... `rn-1 = f en rn`



```
let fold_right f lst b =  
  let rec foldf b = function  
    | [] -> b  
    | h :: t -> f h (foldf b t)  
  in foldf b lst
```

`fold_right` prelucrează elementele de la coadă, nu e *tail-recursive*.

## Exemplu: filtrarea unei liste

```
List.filter : ('a -> bool) -> 'a list -> 'a list
```

```
List.filter f [a1; a2; ...; an] :  
  lista elementelor ak pentru care f ak e adevărată
```

```
let filter f =  
  let rec filt1 = function  
    | [] -> []  
    | h :: t -> let ft = filt1 t in  
                 if f h then h :: ft else ft  
  in filt1
```

```
List.filter (fun x -> x mod 3 = 0) [1;2;3;4;5;6];;  
- : int list = [3; 6]  
List.filter ((<) 3) [1;2;3;4;5;6];; (* fun x -> 3 < x *)  
- : int list = [4; 5; 6] (* (<) 3 = fun x -> (<) 3 x *)
```

Exercițiu: scrieți `filter` folosind `List.fold_right`

## Exemplu: ciurul lui Eratostene

```
(* lista numerelor de la a la b *)  
let fromto a =  
  let rec from2 r b = if b < a then r else from2 (b::r) (b-1)  
  in from2 []  
  
let nondiv d x = x mod d <> 0  
  
let rec sieve = function  
  | [] -> []  
  | h :: t -> h :: sieve (List.filter (nondiv h) t)  
  
let primes = sieve (fromto 2 10000)
```

# Importanța funcțiilor de parcurgere

*Separă* partea mecanică de cea funcțională

(parcurgerea standard a listei de prelucrarea specifică problemei)

Nu necesită scrierea (repetată) a codului de parcurgere.

Intenția prelucrării poate fi scrisă mai clar (mai direct).

Reduce probabilitatea erorilor la sfârșitul prelucrării (lista vidă)

În multe cazuri, această parcurgere standard poate fi *paralelizată*

Aceste idei au fost preluate *dincolo de limbajele funcționale*

Java 8 introduce interfața `Stream<T>`:

are metode de iterare similare (`map`, `filter`, `reduce`)

permite paralelizarea lor

permite prelucrări cu funcții anonime (lambda expressions)