Logică și structuri discrete

Liste

Casandra Holotescu

casandra@cs.upt.ro

https://tinyurl.com/lecturesLSD

Liste

Listele sunt unul din tipurile care reprezintă *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
- nu pot fi infinite (altă noțiune/alt tip: engl. *stream*)
- diferite de *tuple*:
  - tipuri separate pentru perechi (2), triplete (3), etc.
  - dar permit tipuri diferite pe pozițiile 1, 2, ...

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

accesul la elementele listei e *secvențial* (acces direct doar la primul)
- diferit de *vector/tablou*: *acces direct* (cu indice) la orice element

Lista ca tip recursiv

Listele pot fi *definite recursiv*:

O *listă* e { o *listă vidă* ; un *element* (*capul* listei) urmat de o *listă* (numită *coada* listei) }

Atenție: *coada* listei e o *listă*, NU ~~ultimul element~~.

Definiția e *inductivă*: ea definește toate listele (de un anume tip)
pornind de la cea mai simplă (*cazul de bază*),
exprimând cum construim o listă mai mare dintr-una mai mică
(*pasul inductiv*).

Definiția inductivă a mulțimii tuturor listelor

Fie $List[A]$ mulțimea tuturor listelor cu elemente dintr-o mulțime A

notăm cu $[]$ **lista vidă**, $[] \in List[A]$

și cu :: operatorul de **adăugare** a unui element **la stânga** unei liste, pe prima poziție a listei rezultate

(ex. $1 :: [] = [1]$, $2 :: [1] = [2; 1]$, $3 :: [2; 1] = [3; 2; 1]$)

$List[A]$ e o mulțime inductivă:

- *bază*: $[] \in List[A]$
- *inducția*: $\forall x \in A, \forall L \in List[A] \Rightarrow x :: L \in List[A]$

Constructorii lui $List[A]$:

baza $[]$ (lista vidă)

operația de adăugare la stânga a unui element ::

Lista ca tip recursiv în ML

Cu elementele de limbaj cunoscute, putem defini deja liste:

Un tip *recursiv* cu două *variante*:
- lista *vidă*
- lista *construită* dintr-un *element* și altă *listă*

```
type intlist = Nil | Cons of int * intlist
```

O listă cu trei elemente: `Cons(1, Cons(5, Cons(4, Nil)))`

Am *ales* numele `Nil` și `Cons`, sunt clasice (provin din LISP).

Lista ca tip recursiv în ML

Putem defini lista și ca *tip parametrizat* (cu tipul elementului)

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

(am văzut deja că `'a`, `'b` etc. reprezintă tipuri oarecare)

`Cons("unu", Cons("doi", Nil))` are tipul `string list`

Liste în ML

În ML tipul listă e *predefinit*, cu notații mai concise:
constructorul de listă vidă e `[]`
constructorul cu două argumente e scris `::` ca *operator infix*

Putem scrie deci o listă: `1 :: 2 :: 3 :: []`
`::` e asociativ la dreapta, se poate folosi de mai multe ori.

Operatorul `::` *creează* o *nouă listă* (NU modifică lista dată!)
dintr-un *element* (⇒ *capul* noii liste)
și o *listă* (⇒ *coada* noii liste)

Liste în ML

Mai scurt, scriem valori listă între `[ ]` cu `;` între elemente:

```
[1; 3; 2] (*are tipul int list *)
["o"; "lista"; "de"; "siruri"] (*are tipul string list *)
```

Lista e un tip de date *polimorf* (gr. "mai multe forme"),
mai precis, avem *polimorfism parametric* (introdus în ML, 1975)

câte un tip listă pentru fiecare tip de element, dat ca parametru

`int list`, `string list`, etc.

Potrivirea de tipare (pattern matching)

Un tip cu variante (listă vidă / cap+coadă) e prelucrat prin *tipare* (cu un tipar pentru fiecare variantă).

Prin potrivirea de tipare descompunem un obiect după *structură*, identificând (și numind) *părțile componente*.

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

expresie
cu *tipul* expresiilor din dreapta

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

funcție cu *argument implicit* listă
tipul: *stânga* (listă) -> *dreapta*

Valoarea funcției sau expresiei e cea a lui *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

Potrivirea de tipare (cont.)

Când argumentul listă e ultimul, `function` e mai concis:

```
let f1 x = function (* arg: x si lista *)
  | [] -> x
  | h :: _ -> x + h
```

Altfel, putem folosi `match ... with`

```
let addhd lst1 lst2 = match lst1 with
  | [] -> lst2
  | h :: _ -> h :: lst2
```

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

Funcții predefinite cu liste

Modulul `List` are multe funcții pentru lucrul cu liste.
Le folosim cu numele `List.`*numefuncție*

Funcțiile cele mai simple:

- `List.hd` (*head*) – returnează capul listei
- `List.tl` (*tail*) – returnează coada listei

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

Funcții predefinite cu liste

Funcțiile hd și tl nu sunt definite pentru lista vidă.
(sunt funcții parțiale pe tipul listă)

La apel cu [] ele *generează o excepție*.
(Vom discuta în alt curs despre *tratarea excepțiilor*.)

```
# List.hd [];;
Exception: Failure "hd".
# List.tl [];;
Exception: Failure "tl".
```

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

Folosind potrivirea de tipare, putem scrie *orice* funcție cu liste inclusiv cele pentru cap și coadă:

```
let hd = function
  | [] -> failwith "hd"
  | h :: _ -> h
```

```
let tl = function
  | [] -> failwith "tl" (*exc.*)
  | _ :: t -> t
```

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 (* argument: lista *)
  | [] -> 0
  | _ :: t -> 1 + len t
```

Variantă: în parcurgere, acumulăm rezultatul parțial
$\Rightarrow$ încă un parametru: elementele numărate până acum

```
let rec len2 r = function (* inca un arg: lista *)
  | [] -> r
  | _ :: t -> len2 (r+1) t
let len lst = len2 0 lst
```

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

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

Scrierea cu definiții locale

Funcția `len2` e ajutătoare și nu va fi folosită direct.

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

Putem rescrie:

```
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)

Comparăm cele două variante:

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

adunare la *revenirea* din apel
(calcul pentru rezultatul final)

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

adunare la arg. *înainte* de apel
rezultatul retransmis *neschimbat*

Preferăm varianta 2:
Apelul recursiv e *ultima* operație pe acea ramură (*tail recursion*).
$\Rightarrow$ fiecare apel recursiv returnează *aceeași valoare* ca cel anterior.
Compilatorul poate *optimiza* apelul în *iterație* (ciclu).

Recursivitatea finală e practic la fel de eficientă ca iterația.

Varianta 1 poate eșua pe liste lungi, consumând memorie pe stivă proporțional cu lungimea listei.

Funcții simple: testul de membru

Apare valoarea `x` în listă ?

```
let rec mem x = function (* inca un arg: lista *)
  | [] -> 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 .

Parcurgeri standard pentru liste

E natural să *prelucrăm toate elementele* unei liste.

Distingem trei cazuri principale

1. *Transformăm* fiecare element al listei cu aceeași funcție
   obținem o nouă listă `List.map`
2. *Facem* ceva pentru fiecare element (ex. tipărim)
   (fără a produce vreo valoare ca rezultat) `List.iter`
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).

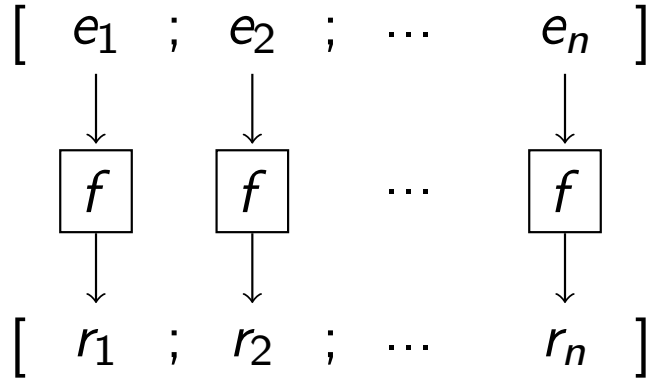
Scriem doar funcția pentru *un pas* de prelucrare (un element),
iar lista e *parcursă automat* de funcțiile standard de iterare.

Transformarea tuturor elementelor dintr-o listă

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

```
List.map f [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



Transformarea tuturor elementelor dintr-o listă

```
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
```

```
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 *)
```

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 (* inca un arg: lista *)
    | [] -> () (* orice functie are valoare, aici () *)
    | h :: t -> f h; iter f t (* a; b are valoarea b *)
```

Iterarea peste toate elementele listei

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 *)
```

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 =    (* f: element -> bool *)
  let rec filt1 = function (* arg: lista *)
    | [] -> []
    | h :: t -> let ft = filt1 t in (* filtreaza coada t *)
                if f h then h :: ft else ft (* ia h daca e bun *)
  in filt1 (* adica: let filter f lst = filt1 lst *)
```

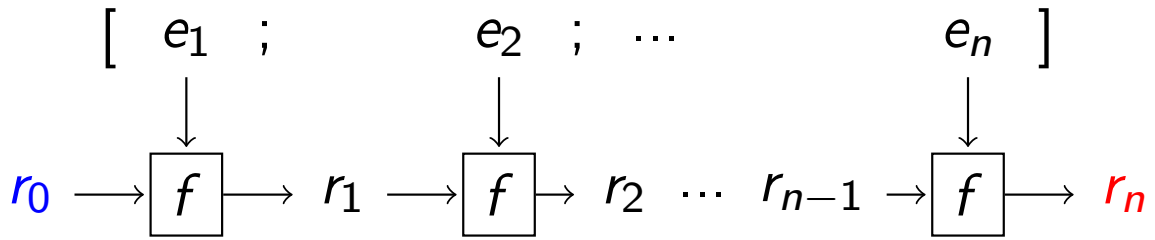
```
 List.filter (fun x -> x mod 3 = 0) [1;2;3;4;5;6];;
- : int list = [3; 6]
List.filter (fun x -> x > 3) [1;2;3;4;5;6];;
- : int list = [4; 5; 6]
```

Exemplu: ciurul lui Eratostene

```
(* lista numerelor de la a la b *)
let fromto a = (* a fix pentru functia interioara *)
  (* adauga ultimul la rez. r, stop la interval vid *)
  let rec fr2 r b = if b < a then r else fr2 (b::r) (b-1)
  in fr2 []     (* arg.1 = [], asteapta arg.2 = b *)

let rec sieve = function
  | [] -> []
  | h :: t -> (* capul e prim, elimina divizorii *)
      h :: sieve (List.filter (fun x -> x mod h <> 0) t)

let primes = sieve (fromto 2 10000)
```

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...)
```



Combinarea elementelor dintr-o listă (de la cap)

```
let fold_left f =
  let rec fold1 a = function (* inca un arg: lista *)
    | [] -> 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:

1. 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
2. valoarea *inițială* de tip `'a`
   (rezultatul pentru lista vidă, și p1 pentru primul apel al lui f)
3. *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
```

Exemple de funcționare pentru `List.fold_left`

Inversarea unei liste:
capul listei rămase 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]
```

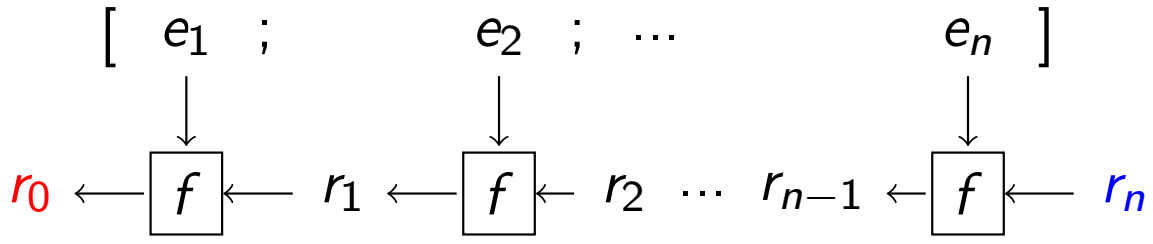
`List.rev` există ca funcție standard pentru liste

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 rezultatul de la dreapta la stânga

`r0 = f e1 r1` ← `r1 = f e2 r2` ← ... r_{n-1} `= f en rn`




```
let fold_right f lst b =
  let rec foldf b = function (* arg. valoare + 1 arg. lista *)
    | [] -> b
    | h :: t -> f h (foldf b t)
  in foldf b lst
```

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

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)

De reținut

Listele sunt cel mai simplu tip *colecție*
  există în multe limbaje, vezi `java.util.Collection`

Lucrul cu *funcții standard de parcurgere*
  cum scriem simplu "fă operația asta pe toată lista"

Prelucrări care au ca parametri *funcții*
  ne permit să indicăm prelucrarea dorită

Lucrul cu liste prin potrivire de *tipare*