3 Liste în ML

O *listă* e un șir ordonat, finit, de elemente de același tip.

În ML, putem scrie valori de tip listă încadrate între paranteze drepte `[` și `]`, cu elementele separate prin punct-virgulă: `[9; 8; 10; 9]` , `["ana"; "are"; "mere"]` .

3.1 Construirea și descompunerea listelor

O listă poate fi definită *recursiv*: e fie *lista vidă*, fie un *element* urmat de o *listă*. În limbajele de programare, listele modelează deobicei această definiție: pentru a ajunge la un anumit element, lista trebuie parcursă pornind de la început (spre deosebire de vectori, care oferă acces *direct* la un element cu un anumit indice).

Pentru a lucra cu liste conform acestei definiții recursive, e nevoie să

- construim o listă vidă: `[]` (indiferent de tipul elementelor)
- construim o listă dintr-o listă existentă și un nou element (care devine *capul* listei, vechea listă devenind *coada* celei noi). Aceasta se face cu *constructorul* `::` cu sintaxa de operator binar infix: *cap* `::` *coada* . De exemplu, `1::[]` e lista `[1]`, iar `1::2::[3;4]` e lista `[1;2;3;4]` .

Atenție! Operatorul `::` produce o listă, nu e nevoie să folosim paranteze drepte în jurul rezultatului. Astfel `1::[2;3]` înseamnă `[1;2;3]` pe când `[1::[2;3]]` înseamnă `[[1;2;3]]`, deci o listă cu un singur element, care e la rândul lui o listă de trei întregi. Deasemenea, `::` *nu poate* lega două liste, ci doar *un element* de o listă (cu elemente de același tip). ~~`[1;2]::[3]`~~ e incorect (nu compilează)!

Exercițiul 1. Scrieți o funcție care returnează lista cifrelor unui număr natural dat ca parametru.

Soluție. Privim numărul recursiv, ca un șir de cifre: fie o singură cifră, fie o cifră (ultima) precedată de alt număr. Cum cifra pe care o extragem dintr-un număr e ultima, iar într-o listă, elementul distinctiv e *primul*, cel mai ușor e să construim lista cifrelor în ordine inversă:

```
let rec revdiglst n = if n < 10 then [n] else (n mod 10) :: revdiglst (n/10)
```

În cazul de bază, se creează o listă cu o cifră (chiar `n`). În cazul recursiv, `revdiglst (n/10)` va produce lista cifrelor primei părți în ordine inversă, la care se adaugă ultima cifră, acum în capul listei. Cum apelul de funcție e mai prioritar decât operatorul `::` nu e nevoie de paranteze în jurul cozii listei.

Putem obține cifrele în ordinea normală dacă acumulăm un rezultat parțial. Scriem o funcție cu *doi* parametri, redefinind exprimarea recursivă a problemei: funcția creează o listă cu cifrele unui număr, știind că din ultima parte a numărului s-a creat deja lista `res` și a mai rămas de prelucrat numărul `n`. Să presupunem că numarul inițial e `1457`, am creat lista `res=[5;7]` și a mai rămas de prelucrat `n=14`. Atunci, ultima cifră a lui `n` trebuie plasată în capul listei `res`, și procesul trebuie continuat recursiv până la epuizarea cifrelor din `n`, când `res` e chiar rezultatul dorit.

```
let rec diglst2 res n = if n < 10 then n::res else diglst2 ((n mod 10)::res) (n/10)
```

Inițial, nu am creat încă nicio listă de cifre, deci funcția trebuie apelată cu valoarea `[]` pentru parametrul `res`. Scriem atunci, așa cum s-a cerut, o funcție cu un singur parametru, și definim `diglst2` local în cadrul acestei funcții, cu sintaxa `let` *ceva* `=` *definiție* `in` *expresie*

```
let diglst =
 let rec diglst2 res n = if n < 10 then n::res else diglst2 ((n mod 10)::res) (n/10)
in diglst2 []
```

Astfel, funcția `diglst` e echivalentă cu `diglst2 []`, ambele luând ca parametru un întreg.

Remarcând faptul că adăugarea ultimei cifre în capul listei e comună pe cele două ramuri, putem refactoriza codul scoțând în afară partea comună:

```
let diglst =
  let rec diglst2 res n =
    let newres = (n mod 10)::res in
    if n < 10 then newres else diglst2 newres (n/10)
  in diglst2 []
```

Definițiile locale cu `let ... = ... in ...` pot fi încuibate deci după nevoi: le folosim pentru a defini nume pentru expresii auxiliare folosite ulterior (posibil în mod repetat).

3.2 Potrivirea de tipare

După definiția recursivă, o listă e fie lista vidă, fie un element (capul) urmat de o altă listă (coada). Pentru a putea scrie funcții care lucrează cu liste, avem nevoie să distingem între aceste două cazuri, iar în al doilea, să *descompunem* o listă în elementele componente (cap și coadă).

În limbajele precum cele funcționale, aceasta se face printr-un mecanism numit *potrivire de tipare* (*pattern matching*), prin care se determină dacă *structura* unei valori (obiect) corespunde unui tipar dat, permițând în acest caz *identificarea* și *numirea* părților componente, și apoi folosirea lor în prelucrări.

Sintaxa potrivirii de tipare e

```
match expr0 with
  | tipar1 -> expr1
  | tipar2 -> expr2
  ...
```

Întreaga construcție e o *expresie*. Dacă valoarea obținută prin evaluarea lui *expr0* se potrivește cu structura indicată de *tipar1*, valoarea întregii expresii e *expr1*; altfel, dacă se potrivește cu structura lui *tipar2*, valoarea rezultantă e dată de *expr2*, etc. Bara verticală `|` pentru prima variantă de tipar e opțională, dar deobicei se scrie, pentru o formatare consistentă a codului.

Cel mai simplu tipar e o valoare constantă: putem astfel scrie pentru negația booleană:

```
let neg b = match b with
  | false -> true
  | true -> false
```

Putem identifica orice alte valori structurate: perechi, n-tuple, liste, tipuri definite de utilizator. Pot fi folosite două sau mai multe tipare cu rezultat comun, separându-le tot cu bara `|`

```
let rec cmmdc a b = match (a, b) with
  | (a, 0) | (0, a) -> a          (* tipare cu o componenta constanta zero *)
  | (a, b) -> cmmdc b (a mod b)   (* orice alt caz *)
```

În scrierea tiparelor sunt valabile următoarele reguli:

- Pe fiecare variantă (ramură), toate numele folosite în stânga lui `->` sunt identificatori *nou definiți* pentru a desemna părțile componente ale tiparului, și domeniul lor de vizibilitate e partea dreaptă (valoarea rezultantă). Chiar folosind nume existente, ele capătă *în cadrul tiparului* un alt înțeles. (Mai sus, identificatorul `a` din tipar e legat de o componentă a perechii, *nu* e parametrul funcției.)
- Un identificator poate apare *o singură dată* într-un tipar, și se potrivește cu orice valoare (cu structură oricât de simplă/complexă).
- Identificatorul special `_` (linia de subliniere) e singurul care poate apare de mai multe ori într-un tipar (fără a implica vreo egalitate între fragmentele structurale pentru care e folosit).
- Potrivirea de tipare se testează pe rând. Deci, pentru tipare cu structură comună, ordinea scrierii contează: o variantă de tipar e aplicabilă doar dacă nu s-au potrivit nicicare din tiparele anterioare.
- Compilatorul avertizează dacă există variante nefolosite sau lipsă (tipare netratate), aceste verificări fiind un mare avantaj al mecanismului de potrivire de tipare (eliminând erorile la rulare).

O scriere simplificată se poate folosi pentru potrivirea de tipare în argumentul unei funcții:

```
let numefct alte_arg = function
  | tipar1 -> expr1
  | tipar2 -> expr2
  ...
```

În acest caz, nu se mai dă nume pentru ultimul (posibil unicul) argument al funcției, deoarece în fiecare variantă, el va fi identificat prin numele date în acele tipare. De exemplu, funcția semn:

```
let sgn = function
  | 0 -> 0
  | x -> if x > 0 then 1 else -1
```

Ordinea variantelor contează aici, pentru că identificatorul `x` se potrivește cu orice tipar; deci, scriind în ordine inversă, și pentru valoarea 0 s-ar returna -1 (dar compilatorul avertizează ca tiparul 0 nu se va activa vreodată).

3.2.1 Tipare pentru liste

În tiparele pentru liste, putem folosi lista vidă și constructorul `::` pentru liste. Cel mai frecvent distingem cele două cazuri: lista vidă sau nu. De exemplu, suma elementelor unei liste de întregi:

```
let rec sumlist = function
  | [] -> 0
  | h :: t -> h + sumlist t
```

Numele `h` și `t` sunt alese de utilizator, sugerând capul listei (*head*) și respectiv coada (*tail*). Putem folosi și tipare mai complicate, de exemplu pentru a afla dacă primele două elemente sunt egale:

```
let equal12 = function
 | e1 :: e2 :: _ when e1 = e2 -> true
 | _ -> false
```

Potrivirea de tipare poate distinge doar *structura* unei valori (sau o constantă), și nu egalitate, deci nu se putea scrie ~~`e :: e :: _ -> true`~~ (compilatorul ar fi semnalat eroare pentru folosirea unui identificator de două ori într-un tipar). Limbajul permite însă adăugarea unei clauze `when` *condiție*, pentru a impune condiții elementelor unui tipar. Linia de subliniere e folosită pentru (sub)tiparele irelevante (coada listei, respectiv al doilea tipar, care acoperă restul cazurilor: listele cu ≤ 1 element, și cele cu primele două elemente distincte. Echivalent se putea scrie mai scurt:

```
let equal12 = function
 | e1 :: e2 :: _ -> e1 = e2
 | _ -> false
```

În acest caz, primul tipar identifică listele cu ≥ 2 elemente (cu rezultatul boolean dat de comparație), iar al doilea tipar, restul (listele cu ≤ 1 element).

Pentru a determina dacă o valoare apare într-o listă, putem scrie că un element e în listă dacă e fie capul listei, fie apare în coada listei (operatorul `||` înseamnă *SAU*):

```
let rec mem x = function
  | [] -> false
  | h :: t -> h = x || mem x t
```

Funcția are *doi* parametri, valoarea căutată (`x`), și lista (pe care folosim potrivirea de tipare cu `function`). Nu am fi putut scrie un tipar ~~`x :: t -> true`~~, pentru că `x` din tipar ar fi un identificator *nou* reprezentând capul listei, fără legătură cu parametrul `x` (la care atunci nu ne-am mai putea referi). Funcția `mem` există și ca funcție definită în modulul standard `List` pentru lucru cu liste.

3.3 Funcții predefinite pentru liste

Funcțiile predefinite pentru liste sunt grupate în modul `List`. Le folosim deci cu numele complet `List.`*numefuncție*. (Dacă am deschide modulul cu directiva `open List` s-ar putea folosi și numele simple, dar se preferă numele complet, pentru a fi mai clar în program unde se lucrează cu liste.)

Două funcții de bază, `List.hd` și `List.tl` permit obținerea capului și cozii unei liste. Dacă argumentul e lista vidă, ele eșuează cu o excepție, după cum putem verifica în interpretorul OCaml:

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

Aceasta înseamnă că înainte de folosirea lor ar trebui să verificăm și să tratăm cazul listei vide. Preferăm lucrul prin potrivire de tipare, deoarece compilatorul verifică automat dacă am tratat toate cazurile; el nu poate detecta însă că nu am comparat o listă cu `[]` înainte de a-i folosi elementele.

Putem implementa noi înșine orice funcție standard din modulul `List`, pentru a înțelege cum funcționează. De exemplu, funcția `length` pentru lungimea unei liste se poate scrie recursiv:

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

Deci, `List.length` nu e o operație atomică, ci necesită parcurgerea întregii liste (ignorând însă valoarea elementelor, după cum arată folosirea lui `_` în tipar). Va fi deci costisitoare pentru liste lungi.

3.4 Recursivitatea finală (tail recursion)

Am folosit recursivitatea pentru a scrie soluții cât mai simple. Trebuie să ne întrebăm și cât de eficiente și utilizabile sunt funcțiile scrise. Să scriem progresia aritmetică cu baza 1 și rația 2 ca șir recurent:

```
let rec arit n = if n = 0 then 1 else 2 + arit (n-1)
```

Apelând în interpretorul OCaml pe un calculator tipic `arit 1000000` (un milion), obținem:
`Stack overflow during evaluation (looping recursion?).`

Observăm că valoarea funcției se poate calcula doar *după* revenirea din apelul recursiv `arit (n-1)`, abia atunci se face suma cu 2. La fiecare apel recursiv, programul trebuie să salveze în memorie locul unde va relua calculul după revenirea din apel, și orice valori (parametri, definiții locale) necesare în continuare. Valorile sunt salvate într-o zonă numită stivă, deoarece ele se vor folosi din nou în ordinea *inversă* în care au fost plasate în memorie (primul apel recursiv e cel din care se revine ultimul). La un număr prea mare de apeluri, memoria din stivă se epuizează, și programul eșuează.

Observăm că pe ramura recursivă, *sigur* vom aduna 2 la rezultatul apelului. Putem anticipa adunarea dând funcției încă un parametru `r`, în care acumulăm valoarea care trebuie adunată. Ajunși la cazul de bază, adunăm la termenul de bază (`1`) valoarea acumulată (`r`) și obținem rezultatul final.

```
let rec arit2 r n = if n = 0 then 1 + r else arit2 (r + 2) (n - 1)
```

Inițial, valoarea rezultatului acumulat e 0 (nu am adunat nimic). Putem scrie deci o funcție `arit1` care apelează `arit2` cu valoarea inițială potrivită (deci `arit1` va fi o funcție cu un parametru):

```
let arit1 =
  let rec arit2 r n = if n = 0 then 1 + r else arit2 (r + 2) (n - 1)
in arit2 0
```

Pe ramura recursivă, rezultatul e returnat direct, nu mai sunt necesare alte calcule. Numim situația recursivitate finală (sau prin revenire; engl. *tail recursion*), pentru că apelul recursiv e *ultima* operație de efectuat pe ramura respectivă. Rezultatul e returnat neschimbat din fiecare apel până la cel inițial, sau (în codul optimizat generat de compilator), chiar direct: nu mai sunt necesare informații pe stivă (adrese de revenire, parametri sau alte valori). Recursivitatea e convertită de compilator în iterație.

Rescriem asemănător lungimea listei, numărând elementele într-un parametru acumulator:

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

Funcția nu mai e limitată de dimensiunea stivei și poate lucra cu argumente mari.

3.5 Parcurgerea cu funcții standard

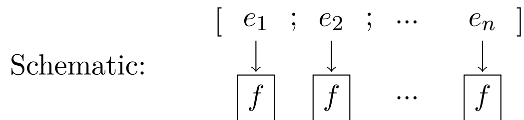
De regulă, listele trebuie parcurse, cu diverse prelucrări pe elemente. Cum prelucrările sunt funcții, și într-un limbaj funcțional putem transmite funcții ca parametri, e natural să folosim funcții standard de parcurgere, la care trebuie doar să specificăm prelucrarea dorită. Fără a mai trebui să scriem codul de parcurgere, avem mai puțin cod, mai simplu și mai ușor de înțeles, și cu mai puține riscuri de eroare.

Distingem trei mari categorii, după cum prelucrarea

- *face* ceva cu fiecare element, fără a returna o valoare (ex. tipărește). Folosim funcția `List.iter` .
- *transformă* fiecare element al listei, generând o nouă listă de aceeași lungime. Folosim `List.map` .
- *combină* valorile din listă (de la cap sau de la coadă). Folosim `List.fold_left` și `List.fold_right`.

3.5.1 `List.iter`

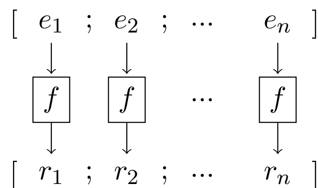
Tipul funcției `List.iter` e `('a -> unit) -> 'a list -> unit`. Aici, `'a` denotă o *variabilă de tip*, care poate fi substituită cu orice tip. Tipul `unit` e un tip cu o singură valoare, notată `()` . E folosit pentru funcțiile al căror scop nu e să producă o valoare utilă, ci un *efect*, cum ar fi de exemplu funcțiile de tipărire. (În C, funcțiile scrise doar pentru efectul lor, fără a returna ceva, au tipul `void`). `List.iter` ia o funcție cu domeniu de definiție arbitrar `'a` și rezultat `unit`, și o listă cu același tip de elemente `'a` și aplică funcția fiecărui element al listei. Ea returnează `unit` (altfel spus, nu produce un rezultat).

Schematic: 

`List.iter Printf.printf "%d "[1;2;3]` va tipări `1 2 3` cu un spațiu (indicat de formatul `"%d "`) după fiecare număr, inclusiv ultimul. `List.iter print_int [1;2;3]` va tipări (fără spații) `123` (`print_int` e o funcție standard cu tipul `int -> unit`, care tipărește un întreg).

3.5.2 `List.map`

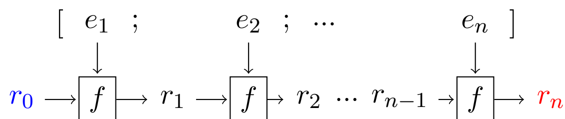
Tipul funcției `List.map` e `('a -> 'b) -> 'a list -> 'b list`. Deci, `List.map` ia ca parametru o funcție arbitrară (cu parametru de tip `'a` și rezultat de tip `'b`), și o listă cu elemente de tip `'a`, aplicând funcția fiecărui element al listei, și returnând lista rezultatelor (o listă de aceeași lungime cu cea inițială, cu elemente de tip `'b`. Schematic (notând cu f funcția aplicată):



Spre exemplu, `List.map (fun x -> x+1) [1;2;3]` sau, echivalent, `List.map ((+) 1) [1;2;3]` produce lista `[2;3;4]`. `List.map String.length ["ana"; "are"; "mere"]` produce lista `[3;3;4]` (funcția `length` din modulul `String` returnează lungimea unui șir).

3.5.3 `List.fold_left` și `List.fold_right`

Funcțiile `List.fold_left` și `List.fold_right` sunt cele mai versatile funcții pentru lucru cu liste. Ele permit calculul unei valori de tip arbitrar din *toate* elementele listei, pornind de la o valoare inițială și aplicând la fiecare pas o funcție dată elementului curent și rezultatului obținut până în acel moment. `List.fold_left` are tipul `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`, deci are trei parametri: o funcție cu doi parametri, de tip `'a` și `'b` și rezultat de tip `'a`; o valoare inițială de tip `'a`, și o listă de elemente de tip `'b`; rezultatul e de același tip `'a` ca și valoarea inițială. Cu notațiile din schema:

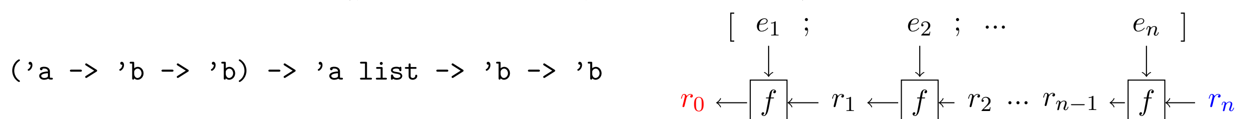


`List.fold_left` calculează succesiv valorile $r_1 = f(r_0, e_1)$, $r_2 = f(r_1, e_2)$, ..., $r_n = f(r_{n-1}, e_{n-1})$, rezultatul final fiind r_n. De exemplu, suma unei liste de numere se obține simplu luând ca funcție `+`, iar ca valoare inițială `0`: `List.fold_left (+) 0 [1;2;3]` dă rezultatul `6`.

Cum tipul `'a` ale rezultatului, valorilor inițiale și intermediare nu trebuie să fie același cu tipul `'b` al elementelor listei, cu `List.fold_left` se pot face prelucrări variate. De exemplu, inversarea elementelor unei liste se obține cu `List.fold_left (fun r e -> e :: r) []` (aplicată listei). Funcția de prelucrare ia rezultatul parțial `r` (partea de listă inversată) și adaugă elementul curent `e` în fața ei. Astfel, dând ca valoare inițială lista vidă, `List.fold_left (fun r e -> e::r) [] [1;2;3]` calculează valorile intermediare `[1]`, `[2;1]`, `[3;2;1]`, returnând ultima ca rezultat. După cum se vede și din tipul lui `List.fold_left`, funcția dată ca parametru ia ca prim argument rezultatul parțial și ca al doilea argument elementul listei.

`List.fold_right` lucrează asemănător, pornind însă calculul de la coada listei, și cu o ordine diferită pentru parametri (și parametrii funcției de prelucrare), după cum se vede din tipul funcției:

`('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`



De exemplu, `List.map f lst` se poate scrie `List.fold_right (fun e r -> f e :: r) lst []`. `List.fold_right` nu e final-recursivă, deci e de preferat `List.fold_left`. În exemplul de mai sus, lista ar putea fi generată și în ordine inversă cu `List.fold_left`, și apoi inversată.