

Laborator 3, 4

Tipul Listă

O listă este o secvență finită de elemente de același tip. Numărul de elemente poate să fie oricât de mare dar nu poate să fie infinit.

Construirea unei liste

Valori listă

OCaml dispune de un tip listă. Pentru a crea o valoare de tip listă putem scrie secvența de elemente care fac parte din listă separate cu `;` între `[]`:

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

Observăm că dacă scriem în interpretor o valoare listă interpretorul de va afișa (asemănător cu comportamentul pentru alte tipuri) tipul valorii și valoarea. Tipul valorii pentru lista pe care am introdus-o mai sus este `int list`. La modul general, tipul unei liste este format din tipul elementului urmat de cuvântul `list`.

Alte exemple de liste:

Listă de numere:

```
# [1.;2.;3.];;  
- : float list = [1.; 2.; 3.]
```

Listă de șiruri de caractere:

```
# ["aa";"bb";"cc"];;  
- : string list = ["aa"; "bb"; "cc"]
```

Listă de funcții:

```
# [ (fun x -> x); (fun x -> x * x); (fun x -> x * x * x)];;  
- : (int -> int) list = [<fun>; <fun>; <fun>]
```

Listă de listă de întregi

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

După cum vedem din exemplele de mai sus, tipul elementelor din listă poate să fie orice alt tip OCaml, inclusiv funcții, și chiar o altă listă.

Dacă elementele unei liste nu au același tip vom primi o eroare din parte interpretorului:

```
# [1;1.];;  
Characters 3-5:  
[1;1.];;  
  ^^  
Error: This expression has type float but an expression was expected of type  
      int
```

Lista vidă

În exemplele de mai sus, listele au conținut cel puțin un element. Putem însă să definim și o listă goală (lista vidă), dacă nu punem nici un element între `[]`:

```
# [];;  
- : 'a list = []
```

Observăm că tipul elementului pentru lista vidă este un tip generic (`'a`). Dat fiind faptul că lista nu conține nici un element, tipul elementului nu poate să fie determinat putând fi orice tip. Acest comportament face lista vidă compatibilă cu orice alt tip de listă:

```
let listOrEmpty x = if x > 0 then [1;2;3] else [];;  
val listOrEmpty : int -> int list = <fun>
```

În exemplul de mai sus avem o funcție care returnează fie o listă cu 3 întregi (dacă `x > 0`) fie o listă vidă (dacă `x < 0`). Observăm că tipul rezultatului acestei funcții va fi dat de lista de după `then` din `if`. Pe ramura `then` avem o listă cu elemente de un tip concret (tipul întreg), pe ramura `else` avem lista vidă care este o listă cu elemente de tip oarecare. Pentru a reconcilia aceste două tipuri (ramurile unui `if` trebuie să returneze același tip de valoare) compilatorul va decide că în acest caz tipul generic al listei vide trebuie să fie înlocuit cu tipul `int`, lista vidă devenind astfel o listă vidă de `int`.

Operatorul `::`

Operatorul `::` permite adăugarea în fața unei liste a unui element nou:

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

În exemplul de mai sus am adăugat elementul cu valoarea `1` în fața listei `[2;3]` și am obținut lista `[1;2;3]`.

Primul operand pentru operatorul `::` trebuie să fie un element al listei iar al doilea operand trebuie să fie o listă, fie o listă cu elemente de același tip ca primul operand, fie lista vidă.

Următoarele apeluri ale operatorului `::` sunt invalide:

```
# 1. :: [2];;  
Characters 7-8:  
  1. :: [2];;  
    ^  
Error: This expression has type int but an expression was expected of type  
      float
```

Elementul care se adaugă este de tip real, dar lista conține elemente de tip `int`

```
# [1]::[2];;  
Characters 6-7:  
  [1]::[2];;  
    ^  
Error: This expression has type int but an expression was expected of type  
      int list
```

Primul operand este o listă de întregi, și îl adăugăm într-o listă de întregi. Operatorul `::` nu va face concatenarea celor două liste, el poate face doar adăugarea unui element de același tip la începutul listei.

Operatorul `::` este asociativ la dreapta deci o expresie de forma

```
1 :: 2 :: 3 :: []
```

Va fi interpretată ca

```
1 :: (2 :: (3 :: []))
```

Dacă operatorul ar fi fost asociativ la stânga expresia ar fi fost interpretată ca

```
((1 :: 2) :: 3) :: []
```

și nu ar fi fost validă (deoarece încercăm să adăugăm un număr la un număr nu o listă - `1 :: 2`).

Construirea recursivă a unei liste

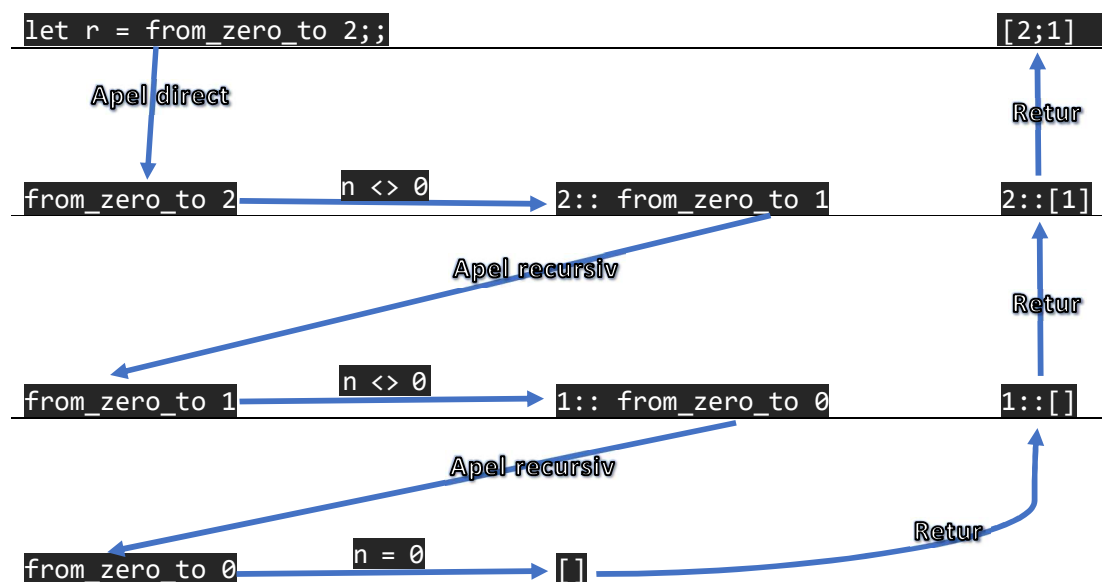
Folosind o funcție recursivă și operatorul `::` putem să construim liste. Spre exemplu să rezolvăm următoarea problemă: Scrieți o funcție care să accepte un argument întreg și să creeze o listă cu toate numerele de la 1 la numărul pasat ca argument.

```
# let rec from_zero_to n = match n with
| 0 -> []
| n -> n :: from_zero_to (n-1)

let r = from_zero_to 10;;
val from_zero_to : int -> int list = <fun>
val r : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
```

Dacă `n` este `0` vom returna lista vidă `[]`. Dacă este alt număr vom adăuga `n` în fața listei returnată de apelul recursiv `from_zero_to (n-1)`.

Să vedem cum se execută apelul `from_zero_to 2`:



Observăm că datorită modului în care se face apelul recursiv valoarea care va ajunge pe prima poziție este valoarea cea mai mare.

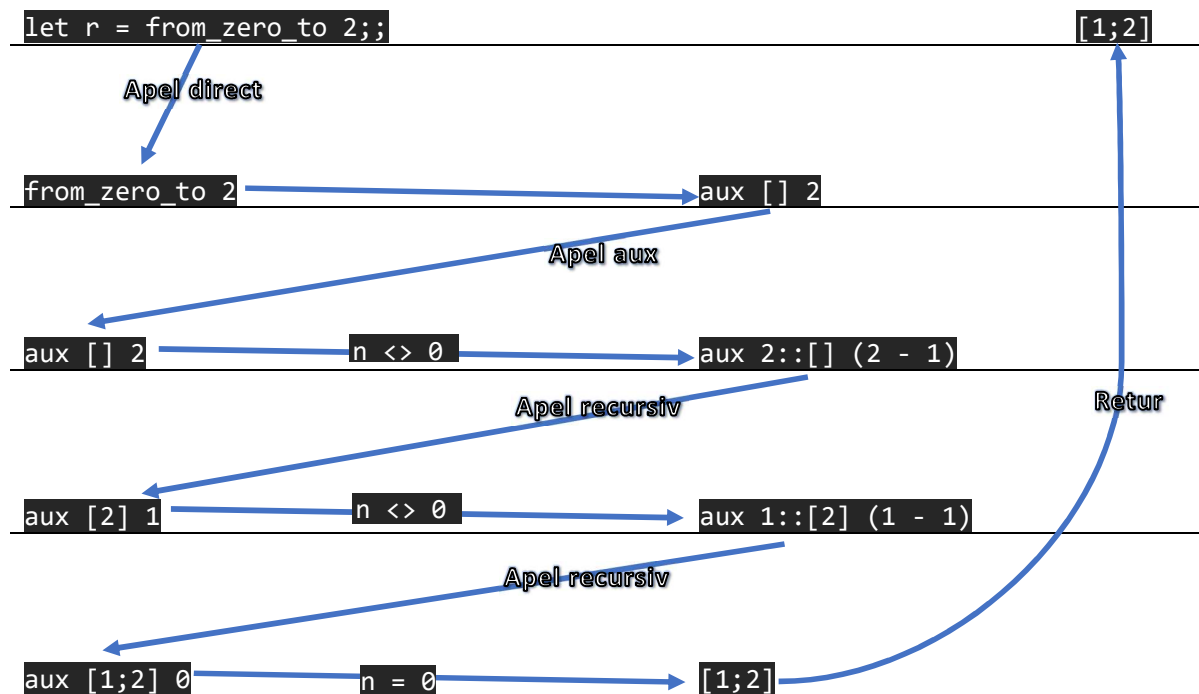
Să vedem și o variantă final recursivă a acestei funcții:

```
# let from_zero_to n =
  let rec aux part n = match n with
    | 0 -> part
    | n -> aux (n::part) (n-1)
  in aux [] n

let r = from_zero_to 10;;
val from_zero_to : int -> int list = <fun>
val r : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Pentru varianta final recursivă parametrul care reține rezultatul parțial va fi o listă (lista care reprezintă valorile acumulate până atunci în listă). Lista parțială va porni de la lista vidă (la început nu am acumulat nimic în listă).

Să vedem cum se execută această funcție pentru apelul `from_zero_to 2`:



Potrivire de tipare/Pattern matching pentru liste

Având o listă vom dori să accesăm elemente ale listei. O metodă de a face acest lucru este folosind potrivirea de tipare.

Tiparul listă vidă

Cel mai simplu tipar este acela care se potrivește cu lista vidă, reprezentat prin însăși lista vidă. Să definim funcția `isEmpty`, care va returna `true` dacă lista pasată ca parametru este goală, și `false` dacă lista nu este goală.

```
# let isEmpty l = match l with
| [] -> true
| _ -> false
let r = isEmpty [] (* Va fi true, lista e goală *)
let r1 = isEmpty [1;2];; (* Va fi false lista nu e goală *)
val isEmpty : 'a list -> bool = <fun>
val r : bool = true
val r1 : bool = false
```

Tipare cu operatorul ::

Operatorul :: poate să apară în tipar, cu semnificația, potrivește tiparul cu o listă care începe cu valoarea din stânga și se termină cu lista din dreapta.

Să scriem o funcție care returnează adevărat dacă lista începe cu valoarea **1**:

```
# let startOne l = match l with
| 1::_ -> true
| _ -> false
let r = startOne [] (* Va fi false nu incepe cu 1 *)
let r1 = startOne [2;3] (* Va fi false nu incepe cu 1 *)
let r2 = startOne [1;2];; (* Va fi true, lista începe cu 1 *)
val startOne : int list -> bool = <fun>
val r : bool = false
val r1 : bool = false
val r2 : bool = true
```

Tiparul **1::** se va potrivi cu o listă care începe cu elementul 1, și va avea orice listă după acest element (tiparul orice **_**).

Asemănător putem scrie un tipar pentru o listă care are orice element la începutul listei, și va avea după acest element o listă **[1]** (lista cu un singur element de valoare 1)

```
# let endOne l = match l with
| _::[1] -> true
| _ -> false
let r = endOne [] (* Va fi false nu incepe cu 1 *)

(* Va fi false, lista se termină cu, dar tiparul dictează ca după primul element
sa vină imediat lista [1], și în acest exemplu urmează lista [2;1]*)
let r1 = endOne [3;2;1]
let r2 = endOne [2;1];; (* Va fi true, conține un 2 urmat de lista [1] *)
val endOne : int list -> bool = <fun>
val r : bool = false
val r1 : bool = false
val r2 : bool = true
```

Variabile în tipar

La fel cum puteam să definim variabile noi în tipare pentru numere, putem să definim variabile noi și în tiparele pentru listă. Aceste variabile pot să apară în orice loc am folosit o valoare în tiparele anterioare și ne vor ajuta să extragem din listă valorile de pe pozițiile respective.

Spre exemplu putem defini o funcție care returnează primul element al unei liste sau care ridică o eroare (numită excepție) dacă nu există elemente în listă folosind funcția `failwith`.

```
# let head l = match l with
| h::_ -> h
| [] -> failwith "Nu sunt elemente"
let h = head [1;2;3];; (* h va fi 1 *)
val head : 'a list -> 'a = <fun>
val h : int = 1
(* h2 nu va avea nici o valoare deoarece head va ridica o excepție
   | oprind executia programului *)
# let h2 = head [];;
Exception: Failure "Nu sunt elemente"
```

Tiparul `h :: _` se va potrivi cu orice listă care are cel puțin un element, urmat de orice (adică fie o listă vidă, fie o listă cu alte elemente). Valoarea primul element va fi pusă în noua variabilă `h`.

Variabila `h` va fi primul element al listei, **deci va avea același tip ca un element din listă**. Spre exemplu pentru o listă de numere întregi, `h` va fi un întreg.

Notă: Există o funcție predefinită `List.hd` care face exact ce face și funcția pe care am definit-o mai sus. `hd`, este o prescurtare pentru `head`, adică capul listei.

Asemănător putem să definim o variabilă în care se va pune restul listei (toate elementele mai puțin primul). Să definim o funcție care returnează o listă care conține toate elementele mai puțin primul element al unei liste pasate ca parametru:

```
# let tail l = match l with
| _::t -> t
| [] -> failwith "Nu sunt elemente"
let t = tail [1;2;3];; (* h va fi [2;3] *)
val tail : 'a list -> 'a list = <fun>
val t : int list = [2; 3]
# (* t2 nu va avea nici o valoare deoarece tail va ridica o excepție
   | oprind executia programului *)
let t2 = tail [];;
Exception: Failure "Nu sunt elemente".
```

Tiparul `_ :: t` se va potrivi cu o listă care are orice element pe prima poziție (tiparul `[]`) urmat de o listă care va fi reținută în variabila `t`.

Variabila `t` va fi o listă care va conține restul elementelor din lista originală, **deci va avea același tip ca lista originală**. Spre exemplu pentru o listă de numere întregi, `t` va fi o listă de întregi.

Notă: Există o funcție predefinită `List.tl` care face exact ce face și funcția pe care am definit-o mai sus. `tl`, este o prescurtare pentru `tail`, adică coada listei.

Putem pune variabile pe ambele poziții într-un tipar cu operatorul `::`. Spre exemplu să scriem o funcție care acceptă o listă de întregi și adună 1 la primul element:

Tipare complexe

Exerciții

1. Scrieți o funcție care returnează `true` dacă lista conține un singur element și returnează `false` altfel
2. Scrieți o funcție care returnează numărul elementelor pentru o listă cu mai puțin de 3 elemente și -1 dacă lista are mai multe elemente (fără recursivitate)

Parcurgerea recursivă

Folosind potrivirea de tipare într-o funcție recursivă putem să parcurgem o listă. În fiecare pas de execuție vom potrivi lista cu un tipar `h::t`. Elementul din capul listei (care va fi pus în variabila `h`) va fi procesat în apelul curent. Restul elementelor din listă (care vor fi puse în variabila `t`) vor fi procesate în apelul recursiv.

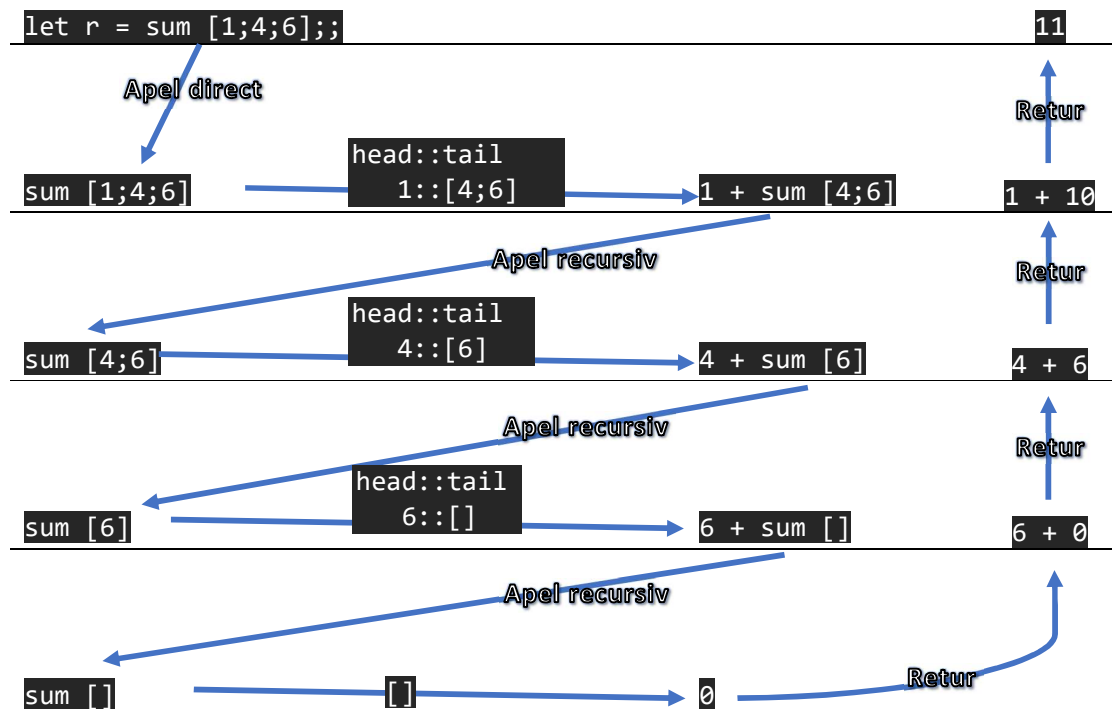
Să vedem cum putem să rezolvăm următoarea problemă: Scrieți o funcție care primește ca parametru o funcție și returnează suma elementelor din listă.

```
# let rec sum lista = match lista with
| [] -> 0
| head :: tail -> head + sum tail

let r = sum [1;4;6];;
val sum : int list -> int = <fun>
val r : int = 11
```

Cazul de bază pentru funcția `sum` este lista vidă. Dacă lista este goală, suma va fi `0`. Dacă lista conține cel puțin un element, vom folosi tiparul `head::tail` pentru a extrage primul element din listă. Pentru acest caz rezultatul va fi suma dintre primul element din listă (în variabila `head`) și suma pentru restul de elemente din listă (în variabila `tail`) care va fi calculată prin apelul recursiv `sum tail`.

Să vedem cum se va executa apelul pentru `sum [1;4;6]`.



Asemănător cu suma primelor n numere naturale, unde era preferabil să scriem o variantă final recursivă din motive de performanță, și pentru suma elementelor dintr-o listă putem să scriem o variantă finală recursivă:

Problemă rezolvată

Scrieți o funcție care să filtreze o listă după o condiție dată ca parametru.

Să începem prin rezolvarea unei variante simplificate: Scrieți o funcție care să filtreze numerele pare dintr-o listă.

```
# let rec filter_div_2 lista = match lista with
| [] -> []
| h::t -> if h mod 2 = 0
then h::(filter_div_2 t)
else filter_div_2 t
let r = filter_div_2 [1;2;3;4;5;6];;
val filter_div_2 : int list -> int list = <fun>
val r : int list = [2; 4; 6]
```

Dacă lista este vidă nu avem ce filtra. Rezultatul va fi tot lista vidă.

Dacă lista are cel puțin un element, ea se va potrivi cu tiparul `h::t`.

Dacă primul element este divizibil cu 2 vom returna o listă formată din elementul respectiv (`h`) și restul de elemente (lista `t`) filtrate prin apelul recursiv al funcției `filter_div_2`.

Dacă elementul nu este divizibil cu 2. Returnăm doar restul de elemente (lista `t`) filtrate prin apelul recursiv al funcției `filter_div_2`.

O întrebare comună la acest tip de problemă de filtrare este: „Cum scot elementul din rezultat?” Răspunsul este simplu, nu „îl scot” din rezultat, ci pur și simplu nu îl adaug în rezultat. Rezultatul se

construiește prin returnarea unei noi liste. Când scriem o astfel de funcție noi suntem cei care decidem cum construim lista rezultat. Dacă elementul `h` pe care tocmai l-am separat din listă nu este adăugat la lista returnată ca rezultat folosind operatorul `::` el nu va fi prezent în rezultat.

Putem simplifica puțin sintaxa acestor funcții eliminând expresia `if` și folosind condiții suplimentare la potrivirea de tipare

```
# let rec filter_div_2 lista = match lista with
| [] -> []
| h::t when h mod 2 = 0 -> h::(filter_div_2 t)
| _::t -> filter_div_2 t
let r = filter_div_2 [1;2;3;4;5;6];;
val filter_div_2 : int list -> int list = <fun>
val r : int list = [2; 4; 6]
```

Am eliminat expresia `if` punând condiția direct pe tiparul `h::t`. Dacă condiția nu este îndeplinită lista se va potrivi cu ultimul tipar `_::t`. La acest tipar nu ne mai interesează elementul care este în capul listei și îl potrivim cu `_` (nu ne interesează deoarece dacă ar fi îndeplinit condiția s-ar fi potrivit cu tiparul anterior) .

Să vedem cum se execută funcția pentru apelul `f_div_2 [2;1;4]` (am redenumit funcția din considerente de spațiu)



Acum că avem o funcție care filtrează elementele divizibile cu 2 cum am putea să modificăm această funcție pentru a filtra după orice condiție?

Problema principală este să vedem cum ar putea condiția să nu fie pusă direct în funcție ci să fie pasată ca argument la funcția care face filtrarea.

Modul în care putem reprezenta o condiție este cu o funcție.

Funcția va fi un parametru al funcției de filtrare. Când o vom folosi îi vom pasa ca argument un element al listei. Dacă funcția returnează `true` atunci elementul va fi pus în listă, iar dacă returnează `false` el nu va fi pus în listă.

```
# let rec filter cond lista = match lista with
| [] -> []
| h::t when cond h -> h::(filter cond t)
| _::t -> filter cond t
let r = filter (fun x-> x mod 2 = 0) [2;1;4]
let r2 = filter (fun x-> x mod 2 = 1) [2;1;4];;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
val r : int list = [2; 4]
val r2 : int list = [1]
```

Funcția `filter` este asemănătoare cu funcția `filter_div_2` scrisă anterior, dar are un parametru în plus, parametru `cond`. Acest parametru va fi funcția care va specifica care să fie elementele filtrate în lista rezultat.

Avantajul acestei metode este faptul că putem refolosi aceeași funcție de filtrare indiferent de condiția pe care vrem să o folosim. În exemplul de mai sus, apelăm funcția `filter` cu două funcții de filtrare una care va returna `true` pentru numerele pare și alta care va returna `true` pentru numerele impare.

Confuzii comune

Pasarea funcțiilor ca parametru este o strategie foarte puternică care ne permite să refolosim același algoritm particularizând doar anumite aspecte ale algoritmului. Această strategie poate însă crea confuzii la început.

De unde știe compilatorul că parametrul `cond` este funcție ?

Compilatorul va ști că `cond` este o funcție din modul în care folosim acest parametru. `cond` este folosit în expresia `... when cond h -> ...`.

`cond h` este un apel de funcție, nu există nici o altă interpretare validă posibilă pentru această expresie. De aici compilatorul deduce că `cond` trebuie să fie o funcție, care este apelată cu argumentul `h`. Dat fiind faptul că argumentul pasat este `h`, adică un element din listă, `cond` va fi o funcție care acceptă ca argument același tip ca un element al listei.

`cond h` este folosit în clauza `when` a unui tipar. Expresiile din astfel de clauze trebuie să aibă tipul `boolean` (`true/false`). De aici deducem că valoarea returnată de `cond` trebuie să fie de tip `boolean`.

Trebuie neapărat funcția să fie parametru ? Nu e suficient să o declar înainte ? Ca de exemplu:

```
let cond x = x mod 2 = 0
let rec filter lista = match lista with
| [] -> []
| h::t when cond h -> h::(filter t)
| _::t -> filter t
let r = filter [2;1;4]
```

Implementarea de mai sus nu ne permite să refolosim funcția `filter` cu diferite condiții. Condiția este mutată într-o funcție dar această funcție este legată direct de funcția `filter`. Nu putem în același program să apelăm funcția `filter` și pentru filtrarea de numere pare și de numere impare. Singurul mod de a schimba condiția este să modificăm codul lui `filter` (pentru a schimba funcția folosită) sau să modificăm codul funcției `cond` (pentru a schimba direct condiția)

Funcția parametru trebuie să aibă același nume cu o funcție existentă ? Ca de exemplu:

```
let par x = x mod 2 = 0
let rec filter par lista = match lista with
| [] -> []
| h::t when par h -> h::(filter par t)
| _::t -> filter par t
let r = filter par [2;1;4]
```

Funcția **filter** de mai sus are un parametru denumit **par**. Există și o funcție definită înainte de **filter** denumită **par**. **Nu există nici o legătură** între funcția **par** și parametrul **par**. Funcția **filter** va fi apelată pe ultima linie și aici parametrul **par** va lua ca valoare funcția **par**.

Cu toate programul funcționează, coincidența de nume creează confuzii. Mai rău numele parametrului nu este de loc sugestiv. Este mai bine să dăm un nume logic acestui parametru care preferabil va fi diferit de numele altor funcții definite.

Parcurgerea cu funcții

Problema rezolvată de mai sus face filtrarea unei liste după orice condiție (pasată ca parametru). Funcția aceasta este suficient de utilă pentru a fi inclusă în biblioteca OCaml. Funcțiile care lucrează cu funcții sunt grupate în modulul **List**. În continuare vom vedea funcția **filter** și alte funcții care sunt folosite în mod comun pentru a face operații pe liste.

List.filter

Funcția **List.filter** ne permite să filtrăm o listă. Funcția va accepta ca parametru o funcție care va decide dacă un anumit element va fi în lista rezultat sau nu.

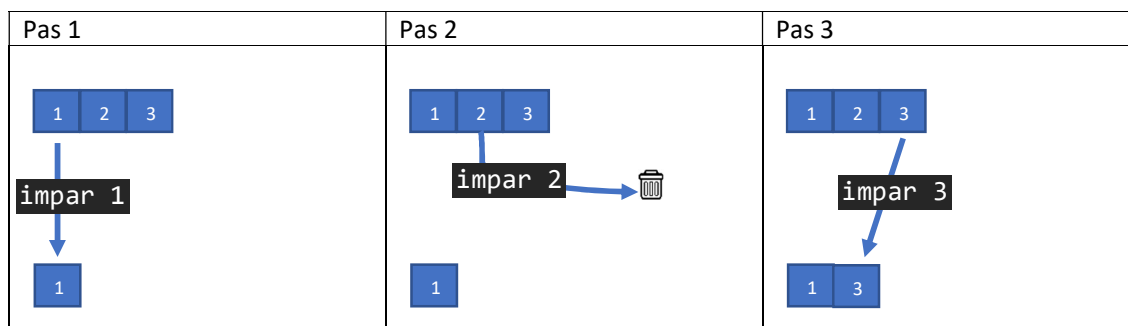
Ex:

```
# let r = List.filter (fun x -> x mod 2 = 1) [1;2;3];;
val r : int list = [1; 3]
```

Funcția argument va fi apelată pentru fiecare element. Dacă funcția returnează **true** elementul va fi inclus în rezultat, dacă returnează **false** nu va fi inclus în rezultat.

Să vedem cum se execută funcția pentru următorul exemplu:

```
let impar x = x mod 2 = 1
let r = List.filter impar [1;2;3];;
```



List.map

Funcția `List.map` va aplica o funcție dată ca parametru pentru toate elementele listei. Rezultatul funcției va fi o listă cu aceeași lungime unde fiecare element este rezultatul apelului funcției parametru pentru elementul de la aceeași poziție.

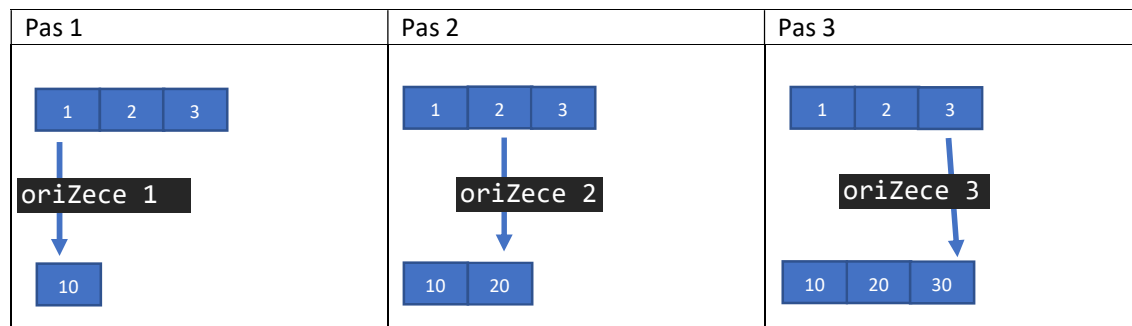
Ex: Fiecare element al listei va fi înmulțit cu 10:

```
# let r = List.map (fun x -> x * 10) [1;2;3];;  
val r : int list = [10; 20; 30]
```

Funcția pasată ca parametru va fi apelată cu fiecare element al listei, deci pentru exemplul de mai sus, funcția va fi apelată de 3 ori, iar parametru `x` va lua pe rând valorile `1`, `2` și `3`.

Să vedem cum se execută funcția pentru următorul exemplu:

```
# let oriZece x = x * 10  
let r = List.map oriZece [1;2;3];;  
val r : int list = [10; 20; 30]
```



List.fold_left

Funcția `fold_left` (împreună cu funcția asemănătoare `fold_right`) este cea mai flexibilă funcție din punct de vedere al aplicațiilor posibile.

Funcția ne permite să distilăm (sau să împăturim, de unde și numele fold) întreaga listă într-o singură valoare.

```
List.fold_left functia_care_face_distilarea valoarea_inițială lista
```

Modul în care se face această distilare este dat de funcția pasată ca parametru. Funcția parametru are 2 parametri, primul parametru va fi valoarea distilată până în momentul respectiv, iar al doilea parametru este un element al listei.

Funcția parametru va fi apelată cu fiecare element al listei (pe post de al doilea parametru). Primul parametru va avea valoarea returnată de apelul anterior. Pentru primul apel valoarea primului parametru este dată de o valoare inițială pasată ca al doilea argument la funcția `List.fold_left`.

Modul în care se va apela funcția parametru va fi următorul:

```

f(vinit, list0) = v0
f(v0, list1) = v1
...
f(vn-2, listn-1) = vn-1
f(vn-1, listn) = vn = rezultat final

```

Să luăm un exemplu pentru a vedea utilitatea acestei funcții. Să calculăm suma numerelor dintr-o listă:

```

# let r = List.fold_left (fun sum_partiala element -> sum_partiala + element) 0 [4;5;6];;
val r : int = 15

```

Să vedem cum se va executa acest apel:

```

# let sum sum_partiala element = sum_partiala + element
  let r = List.fold_left sum 0 [4;5;6];;
val r : int = 15

```

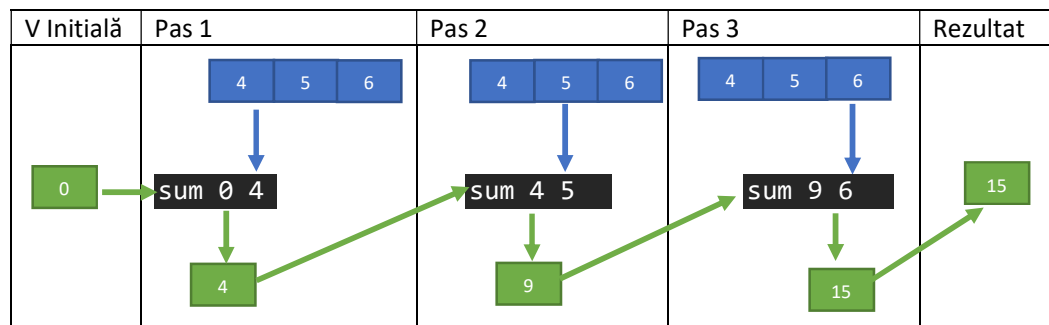
Valoarea inițială pentru parametrul `sum_partiala` va fi `0` (deoarece acesta este valoarea specificată ca al doilea argument pentru `List.fold_left`)

```

sum(vinit = 0, list0 = 4) = 0 + 4 = 4 = v0
sum(v0 = 4, list1 = 5) = 4 + 5 = 9 = v1
sum(v1 = 9, list2 = 6) = 9 + 6 = 15 = v2 = rezultat final

```

Sau folosind o diagramă pentru aceeași operație



Notă: Dat fiind faptul că funcția pasată ca argument nu face altceva decât să adune cele două numere care sunt pasate ca argumente, am putea să scriem suma elementelor unei liste folosind faptul că în OCaml operatorii sunt funcții de două argumente:

```

# let r = List.fold_left (+) 0 [4;5;6];;
val r : int = 15

```

List.map implementat cu List.fold_left

`List.fold_left` poate produce orice tip de valoare, inclusiv o altă listă. Astfel putem să îl folosim să implementăm aproape orice funcție care lucrează cu liste din librăria de bază. Putem spre exemplu să implementăm funcția `List.map`, folosind `List.fold_left`.

Să vedem mai întâi cum am putea folosi `List.fold_left` pentru a obține același rezultat ca pentru apelul:

```
# let r = List.map (fun x -> x * 10) [1;2;3];;
val r : int list = [10; 20; 30]
```

Pentru a face acest lucru avem nevoie să decidem care vor fi valoarea inițială și funcția argument care face distilarea rezultatului.

Valoarea inițială ar trebui să fie o valoare de același tip ca și rezultatul, deci tot o listă (de ce, vom discuta în capitoul următor). O valoare care să fie o valoare neutră pentru listă ar putea să fie lista vidă `[]`.

Funcția care face acumularea va trebui să facă înmulțirea cu 10 a fiecărui element primit, și apoi să îl adauge la rezultatul parțial primit ca prim argument. Acest lucru se poate face folosind operatorul `::`.

```
# let r = List.fold_left (fun list_partiala element -> (10 * element)::list_partiala) [] [1;2;3];;
val r : int list = [30; 20; 10]
```

Să vedem cum s-ar executa aceasta apel (vom nota funcția anonimă pasată ca argument cu `f`)

```
f(vinit = [], list0 = 1) = (10 * 1) :: [] = [10] = v0
f(v0 = [10], list1 = 2) = (10 * 2) :: [10] = [20; 10] = v1
f(v1 = [20; 10], list2 = 3) = (10 * 3) :: [20; 10] = [30; 20; 10] = v2 = final
```

Observăm că rezultatul este în ordine inversă față de elementele din lista originală. Putem rezolva această problemă folosind funcția predefinită `List.rev` care inversează o listă dată ca argument, sau putem `List.fold_right` cum vom vedea în capitoul următor. Pentru simplificare vom ignora diferența de ordine în cele ce urmează.

Pentru a apelul de mai sus într-o funcție echivalentă cu `List.map`, trebuie să facem următoarele:

- Trebuie să învelim apelul într-o funcție
- Trebuie să pasăm lista ca parametru (vrem să putem face maparea pentru orice listă nu doar lista `[1;2;3]`)

```
let myMap lista = List.fold_left (fun list_partiala element -> (10 * element)::list_partiala) [] lista;;
```

- Trebuie să pasăm operația care face maparea (în exemplul nostru expresia `10 * e`) ca parametru pentru funcție. Funcția parametru va accepta un element al listei, și va pune în lista rezultat valoarea returnată de această funcție.

```
let myMap mapper lista = List.fold_left (fun list_partiala element -> (mapper element)::list_partiala) [] lista;;
```

Acum putem să apelăm funcția `myMap` la fel ca funcția `List.map`, și ignorând ordinea, obținem același rezultat:

```
# let myMap mapper lista = List.fold_left (fun list_partiala element -> (mapper element)::list_partiala) [] lista
let r = List.map (fun x-> x * 10) [1;2;3];
let r2 = myMap (fun x-> x * 10) [1;2;3];;
val myMap : ('a -> 'b) -> 'a list -> 'b list = <fun>
val r : int list = [10; 20; 30]
val r2 : int list = [30; 20; 10]
```

Asemănător putem implementa funcția `List.filter`, diferența fiind că funcția de acumulare va include sau nu în rezultat elementul curent bazat pe funcția condiție care a fost pasată ca argument.

Cum decid valoarea inițială?

O observație foarte importantă este faptul că rezultatul și valoarea vor fi întotdeauna de același tip. Acest lucru de datorează modului de funcționare a lui `fold_right` fiind o consecință a următoarelor:

- ultimul rezultat al funcției parametru va fi rezultatul final al lui `fold_right`, deci cele două trebuie să aibă același tip
- valoarea returnată de orice apel al funcției parametru trebuie să fie de același tip ca primul argument (acest lucru este determinat de tipul funcției parametru)
- prima valoare a primului parametru al funcției argument va fi valoarea inițială pasată ca al doilea argument la `fold_right`, deci valoarea inițială trebuie să fie de același tip cu primul parametru al funcției argument.

Dată fiind această relație, prima întrebare când folosim `fold_right` trebuie să fie ce tip va avea rezultatul final? Valoarea inițială va același tip cu rezultatul final.

Dacă rezultatul este un număr, ca la suma elementelor din listă, valoarea inițială trebuie să fie un număr

Dacă rezultatul este o listă, ca la implementarea lui `map`, valoarea inițială trebuie să fie și ea tot o listă.

Acum că știm ce tip are valoarea inițială trebuie să vedem în concret ce valoare trebuie să aibă. Valoarea este dependentă de algoritmul pe care dorim să îl implementăm, dar, în general, va trebui să alegem o valoare neutră pentru operația pe care vrem să o facem (o valoare care să nu aibă vre-un impact asupra rezultatului final).

Un alt indiciu în stabilirea valorii inițiale este cazul în care funcția va fi apelată cu lista vidă. Dacă lista este vidă, funcția argument a lui `fold_left` nu va fi apelată niciodată (deoarece nu există nici o valoare posibilă pentru parametrul care reprezintă un element al listei). Astfel rezultatul final va fi chiar valoarea inițială.

Să aplicăm aceste două criterii de alegere a valorii inițiale pentru implementările de sumă și `map` pe care le-am văzut în acest capitol.

Pentru sumă, care ar fi valoarea corectă dacă lista este goală? Care ar fi o valoare neutră care să nu adauge nimic la rezultat? Răspunsul la ambele întrebări nu poate să fie decât `0`.

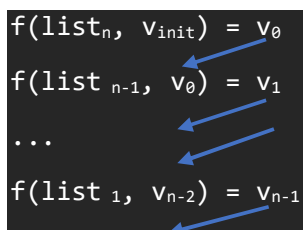
Pentru `map`, care ar fi rezultatul dacă lista este goală? Care ar fi o valoare de tip listă, neutră care să nu adauge nici un element în plus în rezultat? Din nou la ambele întrebări răspunsul nu poate să fie decât lista vidă `[]`.

List.fold_right

Funcția `fold_right` este foarte similară cu funcția `fold_left` diferența fiind ordinea în care funcția argument va primi elementele listei. Dacă la `fold_left` funcția argument va fi apelată prima oară cu valoarea inițială și primul element al listei la `fold_right` primul apel este făcut cu ultimul element al listei și valoarea inițială.

Deci execuția se va realiza în felul următor:

```
f(listn, vinit) = v0
f(listn-1, v0) = v1
...
f(list1, vn-2) = vn-1
```




```
f(list 0, vn-1) = vn = rezultat final
```

Să vedem un apel al acestei funcții, comparativ cu un apel `fold_left`:

```
# let s1 = List.fold_left (fun sum_partiala element -> sum_partiala + element) 0 [1;2;3]
  let s2 = List.fold_right (fun element sum_partiala -> sum_partiala + element) [1;2;3] 0;;
val s1 : int = 6
val s2 : int = 6
```

Observăm că cele două apeluri fac același lucru calculează suma elementelor din lista `[1;2;3]`. Diferențele în modul de apelare constau în:

- Ordinea ultimelor 2 argumente ai funcției `fold_right`, sunt inversați față de `fold_left`, mai întâi avem lista, apoi valoarea inițială la `fold_right`
-

Operatorul `|>`

Operatorul `|>` denumit și operatorul aplicare inversă ne permite să specificăm ultimul parametru al unei funcții înaintea numelui funcției.

Să luăm un exemplu simplu:

```
# let f x = x + 2
  let r1 = 0 |> f
  let r2 = f 0;;
val f : int -> int = <fun>
val r1 : int = 2
val r2 : int = 2
```

Expresiile `0 |> f` și `f 0` fac exact același lucru, apelează funcția `f` cu argumentul `0`.

Putem citi `0 |> f` ca „Trimite `0` la funcția `f`”.

De ce am folosi această metodă de apelare a unei funcții ?

În cazul în care înlănțuim mai multe apeluri (spre exemplu apeluri de funcții care fac procesări de liste) modul normal de scriere a apelurilor devine greu de citit.

Să rezolvăm următoarea problemă: Scrieți o funcție care ia o listă de numere întregi și calculează suma rădăcinii pătrate a numerelor divizibile cu 4.

Pentru a rezolva această problemă trebuie să folosim mai multe operații:

- `List.filter` – pentru a extrage numerele divizibile cu 4
- `List.map` – pentru a transforma numerele întregi în numere reale
- `List.map` – pentru a calcula rădăcina pătrată pentru fiecare element din listă
- `List.fold_left` – pentru a face suma.

Cum ar arăta această funcție scrisă cu apeluri normale:

```
let complicat lista = List.fold_left (fun s e -> s +. e) 0.
  (List.map sqrt
    (List.map float_of_int
      (List.filter (fun x -> x mod 4 = 0) lista)))
```

Această variantă este greu de citit, are multe paranteze care trebuie deschise și închise. De asemenea ordinea în care apar funcțiile în code este exact invers față de modul în care ele se vor executa.

Să vedem aceeași funcție cu operatorul `|>`:

```
let complicat lista = lista
|> List.filter (fun x -> x mod 4 = 0)
|> List.map float_of_int
|> List.map sqrt
|> List.fold_left (fun s e -> s +. e) 0.
```

Ce observăm?

1. Nu mai avem nevoie de paranteze suplimentare
2. Funcțiile apar în ordinea în care vor fi executate

Tipul Tuplă

O **tuplă** este o secvență cu un număr fix de elemente. Elementele pot să fie de același tip sau de tipuri diferite.

Valori tuple

Pentru a crea o valoare de tip tuplă scriem secvența de elemente care fac parte din tuplă separate cu listă separate cu `,` între `()`:

```
# (1, 2, 3);;
- : int * int * int = (1, 2, 3)
```

Observăm că dacă scriem în interpretor o valoare de tip tuplă, interpretorul va afișa tipul valorii și valoarea. Tipul valorii este `int * int * int`, adică o tuplă care poate conține 3 întregi. La modul general tipul unei tuple este format din tipurile elementelor din tuplă separate cu `*`. Notăția `*` provine din faptul că mulțimea definită de un tip tuplă este produsul cartezian al mulțimilor (tipurilor) din care pot face parte elementele tuplei.

Putem defini și tuple care conțin elemente de tipuri diferite:

```
# (1, 2.5, "3");;
- : int * float * string = (1, 2.5, "3")
```

Putem compune tipul tuplă și tipul listă pentru a obține o tuplă de liste

```
# ([1; 2; 3], [1; 2]);;
- : int list * int list = ([1; 2; 3], [1; 2])
```

Sau o listă de tuple

```
# [(1,'a'); (2,'b'); (3, 'c')];;
- : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
```

Tuplele dintr-o listă de tuple trebuie să fie de același tip. Acest lucru înseamnă că:

- Tuplele trebuie să fie de aceeași dimensiune
- Elementele de pe o anumită poziție din tuplă trebuie să fie de același tip.

Ex de liste de tuple greșite:

```
# [(1,'a'); (2,'b',2.)];;
Line 1, characters 10-20:
1 | [(1,'a'); (2,'b',2.)];;
   ^^^^^^^^^
Error: This expression has type 'a * 'b * 'c
      but an expression was expected of type int * char
```

Exemplul de mai sus încearcă să pună în listă o tuplă de dimensiune 2, iar apoi o tuplă de dimensiune 3. Compilatorul va sesiza că tupla cu tipul `'a * 'b * 'c` (tuplă cu 3 elemente de tip oarecare) nu este compatibilă cu tipul primului element din lista de tip `int * char` (tuplă cu două elemente de tip întreg respectiv caracter).

Potrivire de tipare/Pattern matching pentru tuple

Pentru a accesa elemente dintr-o tuplă putem folosi potrivirea de tipare. Un tipar pentru o tuplă trebuie să conțină atâtea elemente în tipar câte elemente trebuie să conțină tupla.

Ex: Scrieți o funcție care acceptă o tuplă de două elemente reprezentând un punct și returnează `true` dacă punctul este originea, și false dacă nu e originea:

```
# let isOrigin p = match p with
| (0, 0) -> true
| (_, _) -> false

let r = isOrigin (0,0)
let r2 = isOrigin (0,1)
;;
val isOrigin : int * int -> bool = <fun>
val r : bool = true
val r2 : bool = false
```

Deoarece tuple de dimensiuni diferite nu sunt compatibile între ele (cum am văzut în exemplul de la lista de tuple) nu putem avea în același `match` tipare pentru tuple cu număr diferit de elemente:

```
#let isOrigin p = match p with
| (0, 0) -> true
| (_, _) -> false
;;
Line 3, characters 4-13:
3 |   | (_, _, _) -> false
    ^^^^^^^^^
Error: This pattern matches values of type 'a * 'b * 'c
      but a pattern was expected which matches values of type int * int
```

Exercițiu propus:

Scrieți o funcție care acceptă ca parametru o tuplă cu două elemente reprezentând un punct și care returnează șiruri de caractere după cum urmează:

- Dacă punctul este `(0,0)` va returna `"Este originea"`
- Dacă punctul este pe abscisă (are al doilea element din tuplă 0) va returna `"Este pe abscisă"`
- Dacă punctul este pe ordonată (are al primul element din tuplă 0) va returna `"Este pe ordonată"`
- Altfel `"Este un punct oarecare"`

Destructurarea unei tuple

De multe ori vrem să aflăm doar valorile elementelor unei tuple. Putem folosi potrivirea de tipare pentru acest lucru:

```
# let pointToString p = match p with
| (x, y) -> Format.sprintf "Punctul de coordonate x = %d y = %d" x y

let r = pointToString (1,2)
;;
val pointToString : int * int -> string = <fun>
val r : string = "Punctul de coordonate x = 1 y = 2"
```

Observăm că expresia `match` conține un singur tipar. Putem simplifica expresia folosind tiparul direct într-o expresie `let`:

```
# let pointToString p =
  let (x, y) = p in
  Format.sprintf "Punctul de coordonate x = %d y = %d" x y

let r = pointToString (1,2)
;;
val pointToString : int * int -> string = <fun>
val r : string = "Punctul de coordonate x = 1 y = 2"
```

Prin `let (x, y) = p` extragem componentele din tupla `p`. Acest fel de atribuire mai poartă numele de destructurare (nu doar în OCaml, [ex](#))

Pe lângă destructurarea prin `let`, mai putem destructura o tuplă și direct în parametrii unei funcții punând tiparul direct în parametrii funcției:

```
# let pointToString (x, y) = Format.sprintf "Punctul de coordonate x = %d y = %d" x y

let r = pointToString (1,2)
;;
val pointToString : int * int -> string = <fun>
val r : string = "Punctul de coordonate x = 1 y = 2"
```

Funcția de mai sus e o funcție cu un parametru de tip tuplă (`int * int`) nu o funcție cu doi parametrii.

Câteva utilizări posibile pentru tuple:

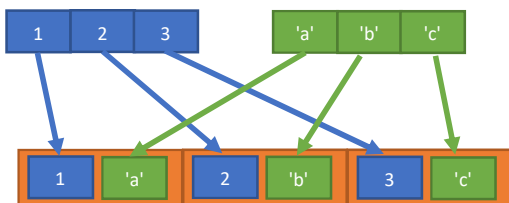
- Când avem date care vrem să le reprezentăm ca într-un singur obiect (de exemplu un punct 2D ca în exemplul de mai sus)
- Când vrem să returnăm dintr-o funcție mai multe valori (de exemplu o funcție care calculează suma și numărul de elemente dintr-o listă)

Funcții cu tuple și liste

List.combine

Funcția `combine` ne permite să luăm două liste de aceeași dimensiune și să le combinăm într-o listă de tuple de dimensiune 2. Tupla de pe poziția n din rezultat va conține pe prima poziție din tuplă elementul n din prima listă, iar pe a doua poziție, elementul n din a doua listă.

```
# let result = List.combine [1;2;3] ['a';'b';'c'];;
val result : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
```



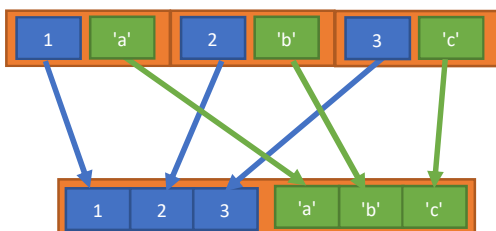
Dacă listele au dimensiuni diferite funcția va ridica o eroare:

```
# let result = List.combine [1;2;3;4] ['a';'b';'c'];;
Exception: Invalid_argument "List.combine".
```

List.split

Funcția **split** este inversa funcției **combine**. Acceptă o listă de tuple de dimensiune 2 și returnează o tuplă cu două liste. Prima listă va conține toate valorile de pe prima poziție a tuplelor din listă, iar a doua listă va conține toate valorile de pe a doua poziție a tuplelor din listă.

```
# let (l1, l2) = List.split [(1, 'a'); (2, 'b'); (3, 'c')];;
val l1 : int list = [1; 2; 3]
val l2 : char list = ['a'; 'b'; 'c']
```



Probleme rezolvate

Produs cartezian