Fundamente de informatică

Prelucrări recursive de liste și expresii

Marius Minea
marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/fi

3 octombrie 2012

Exemplu: inversarea unei liste

Inversăm lista [3;7;5]:

1) separăm capul listei. Va fi ultimul element din lista rezultat.
   lista rămasă: [7; 5] rezultat parțial: [3]
2) separăm capul listei. Se pune la începutul listei rezultat.
   lista rămasă: [5] rezultat parțial: [7; 3]
3) lista rămasă: [] rezultat parțial: [5; 7; 3] (final)

În pasul 1), a crea lista [3] e la fel cu a adăuga 3 în fața listei vide []

```
let rec rev2 rs = function
  | [] -> rs
  | h :: t -> rev2 (h :: rs) t
let rev = rev2 []
```

```
let rev =
  let rec rev2 rs = function
    | [] -> rs
    | h :: t -> rev2 (h :: rs) t
  in rev2 []
```

Varianta a 2-a "ascunde" definiția funcției ajutătoare `rev2` care va fi vizibilă doar în definiția lui `rev`, cu sintaxa

`let` *nume* = *expresie* `in` *expresie*

Exemplu: eliminarea duplicatelor din listă

```
let rec nodup = function
  | h1 :: (h2 :: _ as t) ->
    let rez = nodup t in
    if h1 = h2 then rez else h1 :: rez
  | lst -> lst
```

Putem testa duplicate doar într-o listă cu minim 2 elemente:

tiparul `h1 :: h2 :: _`

când nu ne interesează o valoare, folosim tiparul `_`

Ultimul caz se potrivește pentru liste cu 0 sau 1 element (nu se schimbă).

Cu sintaxa

tipar as *nume*

putem folosim apoi *nume* pentru valoarea care s-a potrivit cu tiparul

Prelucrări iterative pe liste

Pe liste se pot defini funcții generice de prelucrare.
$\Rightarrow$ putem itera prelucrări fără a scrie repetat același cadru

Modulul `List` din ML are astfel de funcții:

```
iter :  ('a -> unit) -> 'a list -> unit
```
`iter f [a1; a2; ...; an]` apelează `f a1; f a2; ... f an; ()`

```
map :  ('a -> 'b) -> 'a list -> 'b list
```
`map [a1; a2; ...; an]` e lista `[f a1; f a2; ... f an]`

```
fold_left :  ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
fold_left f a [b1; b2; ... bn] = f (...f (f a b1) b2...) bn

fold_right :  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
fold_right f [a1; a2;...; an] b = f a1 (f a2 (...(f an b)...))
```

`filter f [a1; a2; ...; an]` : elementele pentru care `f` e adevărată

Implementarea iteratorilor

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

let rec map f = function
    | [] -> []
    | h :: t -> f h :: map f t

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

let rec fold_right f lst b = match lst with
      [] -> b
    | h :: t -> f h (fold_right f t b)

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

Exemple de folosire a iteratorilor

`List.iter print_int [1;2;3]` tipărește 123

`List.map ((+) 2) [3; 7; 4]` are ca rezultat `[5; 9; 6]`

`List.fold_left (+) 0 [3; 7; 4]` face suma elementelor: 14
Putem implementa inversarea `rev` cu `fold_left` :
`let rev = List.fold_left (fun t h -> h :: t) []`

Putem implementa minimul unei liste:

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

Importanța iteratorilor

Separă partea mecanică de cea funcțională
(parcurgerea listei, care e standard, 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)

Recursivitatea prin revenire (tail recursion)

Ordinea operațiilor la `fold_left` și `fold_right` e diferită.

fold_right: `f a1 (f a2 (...(f an b)...))`

primul apel `f a1 ...` are nevoie de o valoare încă necalculată

```
 | h :: t -> f h (fold_right f t b)
```

pe rezultatul dat de instanța apelată mai trebuie un calcul `f h` *rezultat*

e nevoie de înregistrarea pe stivă (valorile proprii ale parametrilor, etc)

fold_left: `f (...f (f a b1) b2...)  bn`

primul apel: `f a b1` care e folosit ulterior

```
 | h :: t -> fold_left f (f a h) t
```

rezultatul returnat de instanța apelată e chiar cel dorit în instanța curentă

fără calcule la revenire, se transmite doar *același* rezultat mai sus

$\Rightarrow$ Recursivitatea e "la coada" prelucrării (tail recursion)

$\Rightarrow$ *Implementare eficientă, transformată automat de compilator în ciclu*

Recursivitatea prin revenire (tail recursion)

Necesită uneori rescrierea prelucrării cu un acumulator

```
let rec fact n =
  if n = 0 then 1 else n * fact (n-1)

let rec fact2 n res =
  if n = 0 then res else fact2 (n-1) (n * res)
```

Exemplu: ciurul lui Eratostene

```
(* lista numerelor de la a la b *)
let rec fromto a b =
  if a > b then [] else a :: fromto (a+1) b

let rec sieve = function
  | [] -> []
  | h :: t -> h :: sieve (List.filter (fun x -> x mod h <> 0) t)

let primes = sieve (fromto 2 1000)
```