(3) Types produits – Fonctionelles sur les listes

Programmation fonctionnelle (LU2IN019)

Licence d'informatique
2023/2024

Jean-Claude Bajard – Mathieu Jaume

Types produits : paires

- les valeurs peuvent être regroupées en paires pour former d'autres valeurs ...

```
# (4, 6.7);;
- : int * float = (4, 6.7)

# (not true, 3 * 2);;
- : bool * int = (false, 6)

# let x = (3 * 4, "ab" ^ "cd");;
val x : int * string = (12, "abcd")
```

- la virgule `,` est ici un constructeur de paire
- si l'expression e_1 est de type t_1 et l'expression e_2 est de type t_2, alors l'expression (e_1, e_2) est de type $t_1 * t_2$
  - $t_1 * t_2$ est un type produit

Types produits : paires

- **syntaxe** : ne pas confondre
  - (e_1, e_2) est une paire
    - ★ e_1 est la première composante de la paire (e_1, e_2)
    - ★ e_2 est la deuxième composante de la paire (e_1, e_2)
  - $(e_1 \ \ e_2)$ est l'application de e_1 sur e_2
    - ★ e_1 est la fonction/l'expression fonctionnelle
    - ★ e_2 est l'argument sur lequel est appliquée la fonction

Types produits : paires

- les composantes d'une paire peuvent être elles-mêmes des paires

```
# ((2 + 1, not true), "ab" ^ "cd");;
- : (int * bool) * string = ((3, false), "abcd")

# let x = (2, false);;
val x : int * bool = (2, false)

# let y = (x, x);;
val y : (int * bool) * (int * bool) = ((2, false), (2, false))
```

- Attention au parenthésage : $(t_1 * t_2) * t_3 \neq t_1 * (t_2 * t_3)$

```
# ((1, 'a'), false) = (1, ('a', false));;
Error : This expression has type int but an expression
was expected of type int * char
```

Accès aux composantes d'une paire

- **fonctions prédéfinies** : utilisation des projections

```
fst : 'a * 'b -> 'a      snd : 'a * 'b -> 'b
```

  - fonctions polymorphes : agissent sur des paires dont les composantes sont de types quelconques

```
# (fst (true || false, 4 - 6));;
- : bool = true

# (snd ((2.5, 666), true));;
- : bool = true

# (fst ((2.5, 666), true));;
- : float * int = 2.5, 666

# (snd (fst ((2.5, 666), true)));;
- : int = 666
```

Accès aux composantes d'une paire

- **liaison filtrante** (`let`)

```
# let (x, y) = (2, true) in
  if y then x else x + 1;;
- : int = 2
# let z = (2, true) in
  let (x, y) = z in
  if y then x else x + 1;;
- : int = 2
```

- **filtrage** (*pattern matching*)
  expression de filtrage (`match ... with ...`)

```
# match (2, true) with
  | (x, y) -> if y then x else x + 1;;
- : int = 2
# let z = (2, true) in
  match z with
  | (x, y) -> if y then x else x + 1;;
    - : int = 2
```

Composantes d'une paire : définitions

- définition des fonctions (prédéfinies) de projections
  - liaison filtrante

```
let fst (c : 'a * 'b) : 'a =
  let (x, _) = c in x

let snd (c : 'a * 'b) : 'b =
  let (_, y) = c in y
```

remarque : comme dans un motif de filtrage, on utilise le caractère _ pour les variables dont on n'a pas besoin

  - liaison filtrante : paramètre de la fonction

```
let fst ((x, _) : 'a * 'b) : 'a = x

let snd ((_, y) : 'a * 'b) : 'b = y
```

Composantes d'une paire : exemple

- fonction qui détermine si les deux composantes d'une paire sont égales

```
eq_comp : 'a * 'a -> bool = <fun>
```

  - liaison filtrante : `let`

```
let eq_comp (c : 'a * 'a) : bool =
  let (x, y) = c in
  x = y
```

  - liaison filtrante : paramètre de la fonction

```
let eq_comp ((x, y) : 'a * 'a) : bool = (x = y)
```

```
# (eq_comp (2,2));;
- : bool = true
# (eq_comp (2,3));;
- : bool = false
# (eq_comp 2 2);;
Error: This function has type 'a * 'a -> bool
       It is applied to too many arguments
```

Composantes d'une paire : exemple

fonction `add_cpl` : $((x_1, x_2), (x_3, x_4)) \mapsto (x_1 + x_2, x_3 + x_4)$

```
let add_cpl (c : (int * int) * (int * int)) : int * int =
  match c with
   | ((x1, x2), (x3, x4)) -> (x1 + x2, x3 + x4)

let add_cpl (((x1,x2),(x3,x4)):((int*int)*(int*int))):int*int =
 (x1 + x2, x3 + x4)
val add_cpl : (int * int) * (int * int) -> int * int = <fun>

# (add_cpl ((1, 2), (3, 4)));;
- : int * int = (3, 7)
```

Types produits / *n*-uplets

- les paires sont des 2-uplets
- les valeurs peuvent être regroupées en paires, triplets ou plus généralement en *n*-uplets

```
# (2. *. 3.14 , 3 < 4 , "O'" ^ "caml");;
- : float * bool * string = (6.28, true, "O'caml)"

# (5,(let s = "ab" in s ^ " " ^ s), not true, (2, true));;
- : int * string * bool * (int * bool)
 = (5, "ab ab", false, (2, true))
```

- la virgule `,` est un **constructeur de *n*-uplets**
  - structures linéaires statiques
- attention au parenthésage des expressions/types produits :

$$\begin{array}{ccccc} (1,\texttt{true},5) & \neq & ((1,\texttt{true}),5) & \neq & (1,(\texttt{true},5)) \\ t_1 * t_2 * t_3 & \neq & (t_1 * t_2) * t_3 & \neq & t_1 * (t_2 * t_3) \end{array}$$

Accès aux composantes d'un *n*-uplet

exemple : accès à la troisième composante d'un triplet

- liaison filtrante

```
let third ((_,_,x) : ('a*'b*'c)) : 'c = x
```

- expression de filtrage

```
let third (t : ('a*'b*'c)) : 'c =
  match t with
  | (_,_,x) -> x
```

```
val third : 'a * 'b * 'c -> 'c = <fun>
# (third (0.5 +. 0.5, 2, "trois"));;
- : string = "trois"
```

n-uplets : paramètres/résultats d'une fonction

fonction calculant le quotient et le reste de la division entière de deux entiers

```
(* quotient et reste de la division de a par b *)
let rec div ((a, b) : (int * int)) : (int * int) =
  if a < b then (0, a) else
    let (q, r) = div (a - b, b) in
    (q + 1, r)

# (div (5, 2));;
- : int * int = (2, 1)
```

- pas d'application partielle car utilisation d'un seul argument (un n-uplet)

```
# (div 5);;
Error: This expression has type int but an expression
       was expected of type int * int
```

n-uplets : paramètres/résultats d'une fonction

pour pouvoir faire une application partielle, il faut utiliser plusieurs arguments

```
let rec div (a : int) (b : int) : (int * int) =
  if a < b then (0, a) else
    let (q, r) = div (a - b) b in
    (q + 1, r)

let div_partial = div 5
val div_partial : int -> (int * int)

# (div 5 2);;                    # (div_partial 2);;
- : int * int = (2, 1)           - : int * int = (2, 1)

# (div 5 4);;                    # (div_partial 4);;
- : int * int = (1, 1)           - : int * int = (1, 1)
```

Ordre supérieur

- un langage de programmation d'ordre supérieur est un langage dans lequel les fonctions sont des valeurs comme les autres
  - elles peuvent être passées en arguments à d'autres fonctions
  - elles peuvent être le résultat d'une fonction

```
let applique_paire (f: 'a -> 'b) ((x,y): 'a*'a) : 'b*'b =
  (f x, f y)

let est_positif (x : int) : bool = (x >= 0)

# (applique_paire est_positif (2, -3));;
- : (bool * bool) = (true, false)
```

- pour utiliser `applique_paire`, il faut définir avant la fonction `f` à appliquer à chaque élément de la paire :
  - `f` est définie dans l'environement courant, même si on ne l'utilise qu'une fois

  On peut éviter cela : définition **anonyme**, définition **locale**

Fonction anonyme, définition locale d'une fonction

- fonction anonyme : `fun (x : t) -> ` e_f
- application d'une fonction anonyme à un argument e_a :

  `((fun (x : t) -> ` e_f`) ` e_a`)`

  - le parenthésage de l'expression fonctionnelle (...) est nécessaire

```
# ((fun (x : int) -> 2 * x) (2 + 3));;
- : int = 10
```

  - lors de l'application, le corps de la fonction e_f est évalué dans l'**environnement courant** augmenté de la liaison (x, v_a)
    - ★ v_a : résultat de l'évaluation de e_a dans l'environnement courant

    l'**environnement de définition d'une fonction anonyme** est l'environnement courant

Fonction anonyme : exemple

```
let applique_paire (f: 'a -> 'b) ((x,y): 'a*'a) : 'b*'b =
  (f x, f y)
```

- *exemple :* avec une fonction **anonyme**

```
# applique_paire (fun (x : int) -> (x >= 0)) (2, -3);;
- : (bool * bool) = (true, false)
```

- on s'autorisera à ne pas donner les types des fonctions anonymes :

```
# applique_paire (fun x -> (x >= 0)) (2, -3);;
- : (bool * bool) = (true, false)
```

Définition locale d'une fonction

- définition locale d'une fonction

```
let f = fun x -> e_f in e
```

```
# let f = fun x -> x + 1 in
  (f (f 4));;
- : int = 6
```

- autre syntaxe, similaire aux déclarations dans l'environement courant

```
let f (x : t) : t' = e_f in e
```

```
# let f (x : int) : int = x + 1 in
  (f (f 4));;
- : int = 6
```

- avec plusieurs arguments

```
let f = fun x y -> e_f in e
```

```
# let g = fun x y -> x + y in
  (g (g 4 1) 1);;
- : int = 6
```

Définition locale d'une fonction : exemple

```
let applique_paire (f: 'a -> 'b) ((x,y): 'a*'a) : 'b*'b =
  (f x, f y)
```

- *exemples :* avec une fonction **locale**

```
# let est_positif : int -> bool = fun x -> (x >= 0) in
  applique_paire est_positif (2, -3);;
- : (bool * bool) = (true, false)

# let est_positif (x : int) : bool = (x >= 0) in
  applique_paire est_positif (2, -3);;
- : (bool * bool) = (true, false)
```

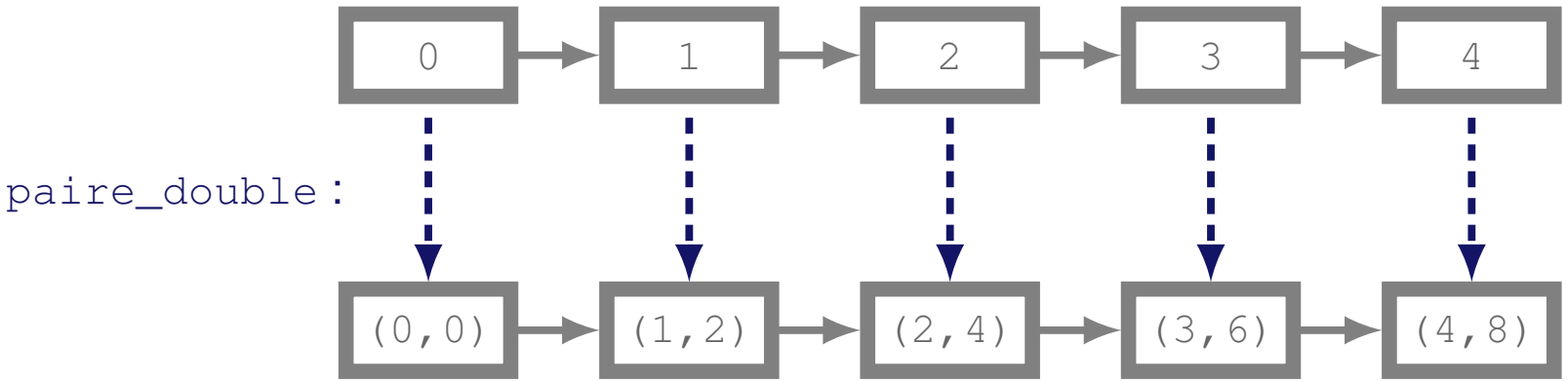
Fonctionnelles sur les listes

- schémas génériques d'itération (traitement/exploration) sur des listes
- `https://ocaml.org/api/List.html`

Schéma d'application : exemple

- appliquer la même fonction sur tous les éléments d'une liste :

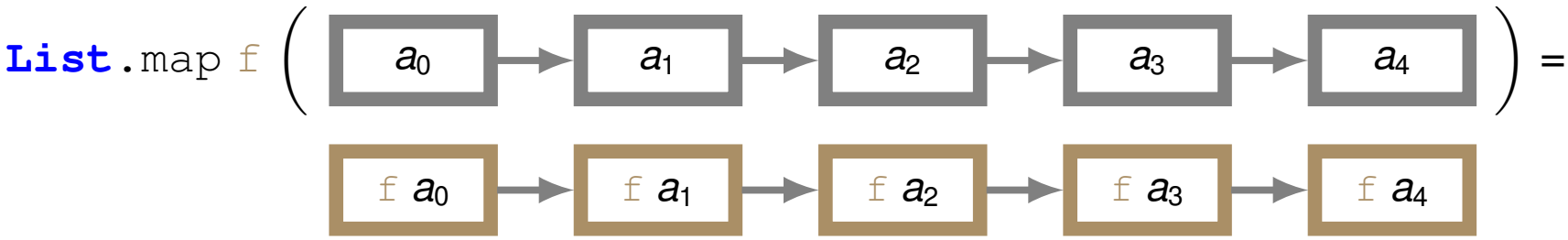
```
let paire_double (x : int) : int * int = (x, 2 * x)
```



- possible grâce à la fonctionnelle `List.map`, qui prend en argument :
  - une fonction `f : 'a -> 'b`
  - une liste d'élément `l : 'a list`

  et qui applique `f` à tous les éléments de `l`, ce qui donne une `'b list`
- *remarque.* `List.map` est une fonction qui prend une autre fonction en argument : c'est une fonction d'**ordre supérieur**

Schéma d'application

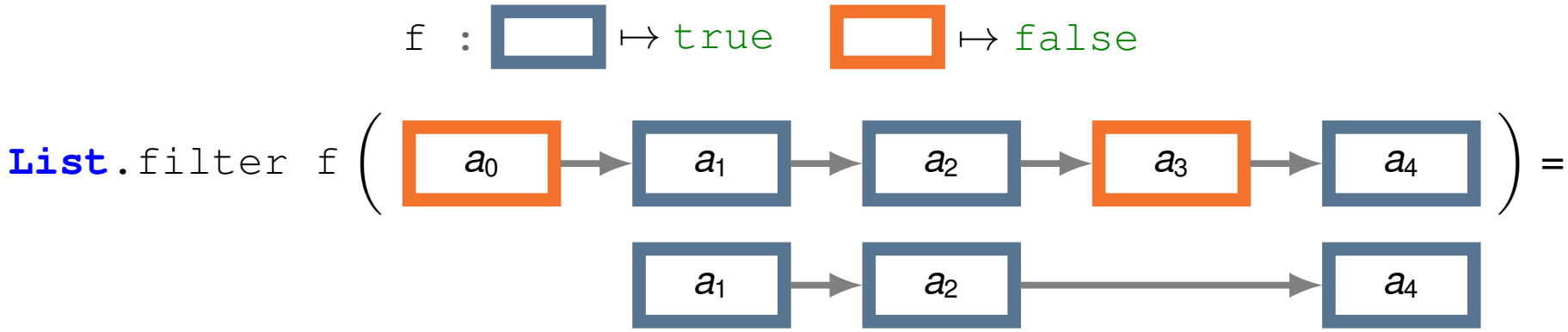


`val map : ('a -> 'b) -> 'a list -> 'b list`
`List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.

```
let rec map (f: 'a -> 'b) (l : 'a list) : 'b list =
  match l with
  | [] -> []
  | a :: t -> (f a) :: (map f t)
```

```
# int_of_char;;
- : char -> int = <fun>
# List.map int_of_char ['a';'z';'A';'Z'];;
- : int list = [97; 122; 65; 90]
```

Schéma de filtrage

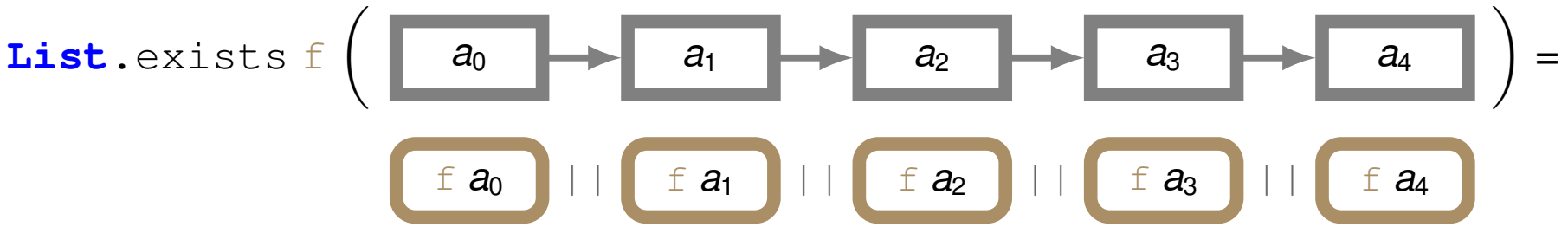


val filter : ('a -> **bool**) -> 'a **list** -> 'a **list**
filter f l returns all the elements of the list l that satisfy the predicate f.
The order of the elements in the input list is preserved.

```
let rec filter (f : 'a -> bool) (l : 'a list) : 'a list  =
  match l with
   | [] -> []
   | a :: t -> if f a then a :: (filter f t) else (filter f t)

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

Schéma d'exploration

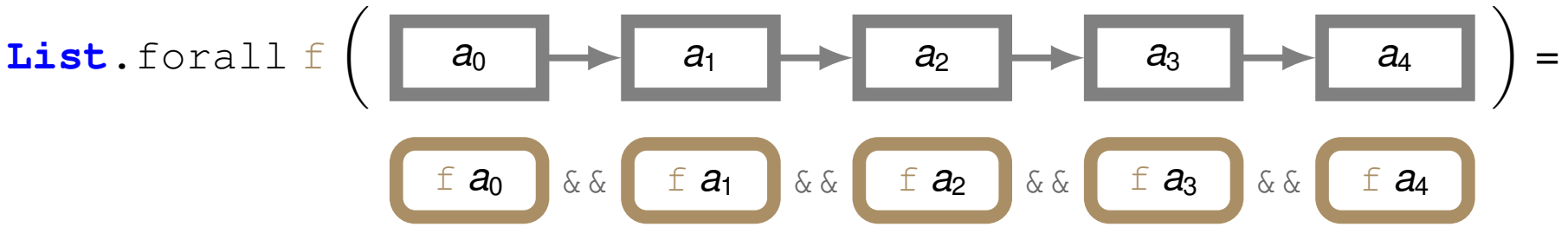


`val exists : ('a -> bool) -> 'a list -> bool`
`exists f [a1; ...; an]` checks if at least one element of the list satisfies the predicate `f`. That is, it returns `(f a1) || (f a2) || ... || (f an)` for a non-empty list and `false` if the list is empty.

```
let rec exists (f: 'a -> bool) (l : 'a list) : bool =
  match l with
  | [] -> false
  | a :: t -> if f a then true else exists f t

# List.exists (fun x -> (x mod 2) = 0) [1;3];;
- : bool = false
# List.exists (fun x -> x > 2) [1;2;3;4];;
- : bool = true
```

Schéma d'exploration

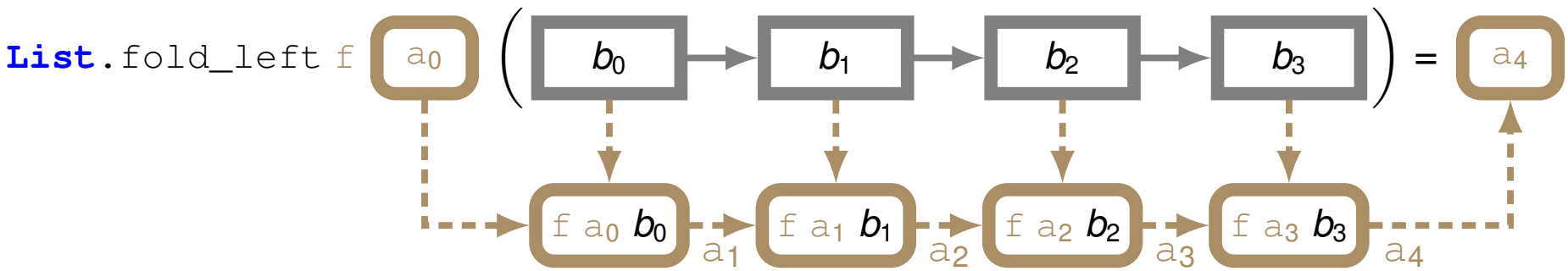


`val for_all : ('a -> bool) -> 'a list -> bool`
`for_all f [a1; ...; an]` checks if all elements of the list satisfy the predicate `f`. That is, it returns `(f a1) && (f a2) && ... && (f an)` for a non-empty list and `true` if the list is empty.

```
let rec for_all (f: 'a -> bool) (l:'a list) : bool =
  match l with
   | [] -> true
   | a :: t -> if f a then for_all f t else false

# List.for_all (fun x -> (x mod 2) = 0) [1;2;3];;
- : bool = false
# List.for_all (fun x -> x > 0) [1;3];;
- : bool = true
```

Schéma d'accumulation




```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_left f a [b0; ...; bn] is
f (... (f (f a b0) b1) ...) bn.
```

```
let rec fold_left (f: 'a -> 'b -> 'a) (a:'a) (l:'b list) : 'a =
  match l with
   | [] -> a
   | b :: t -> fold_left f (f a b) t
```

Schéma d'accumulation : exemple

```
let rec fold_left (f: 'a -> 'b -> 'a) (a:'a) (l:'b list) : 'a =
  match l with
  | [] -> a
  | b :: t -> fold_left f (f a b) t
```

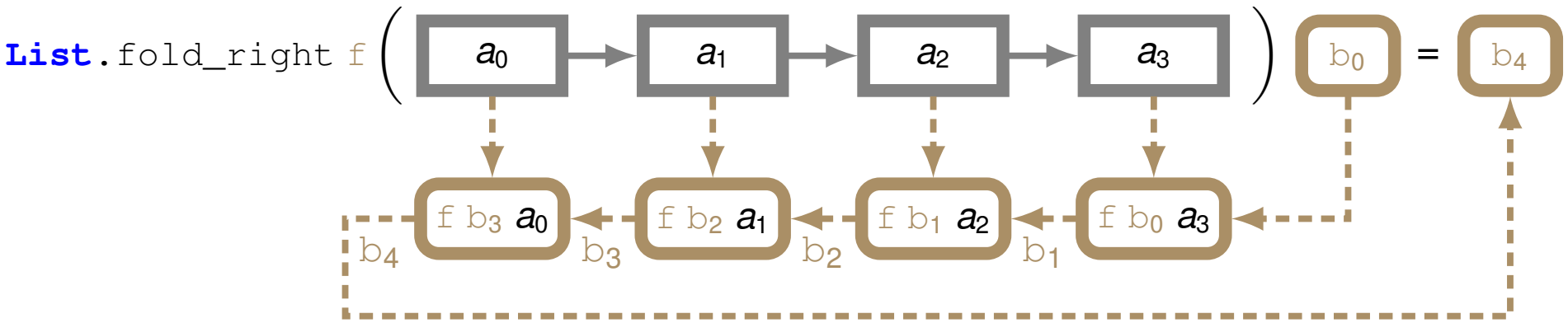
- exemple :

```
# List.fold_left (fun acc x -> acc - x) 0 [1;2;3];;
- : int = -6
```

détails pour `f = (fun acc x -> acc - x)`

```
   List.fold_left f       0  [1;2;3]
=  List.fold_left f  (0 - 1) [2;3]
=  List.fold_left f (-1 - 2) [3]
=  List.fold_left f (-3 - 3) []
=  -6
```

Schéma d'accumulation




```
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.fold_right f [a0; ...; an] b is
f a0 (f a1 (... (f an b) ...)).
```

```
let rec fold_right (f: 'a -> 'b -> 'b) (l:'a list) (b:'b) : 'b =
  match l with
  | [] -> b
  | a :: t -> f a (fold_right f t b)
```

Schéma d'accumulation : exemple

```
let rec fold_right (f: 'a -> 'b -> 'b) (l:'a list) (b:'b) : 'b =
  match l with
  | [] -> b
  | a :: t -> f a (fold_right f t b)
```

- exemple :

```
# List.fold_right  (fun x acc -> x - acc) [1;2;3] 0;;
- : int = 2
```

détails pour `f = (fun x acc -> x - acc)`

```
   List.fold_right f [1;2;3]  0
 = f 1 (List.fold_right f [2;3] 0)
 = f 1 (f 2 (List.fold_right f [3] 0))
 = f 1 (f 2 (f 3 (List.fold_right f [] 0)))
 = f 1 (f 2 (f 3 0))
 = f 1 (f 2 3)
 = f 1 (-1)
 = 2
```

Liste d'associations

liste de couples (*key*, *value*)

`val assoc : 'a -> ('a * 'b) list -> 'b`
`assoc a l` returns the value associated with key `a` in the list of pairs `l`. That is, `assoc a [ ...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Raise **`Not_found`** if there is no value associated with `a` in the list `l`.

```
let rec assoc (c : 'a) (l : ('a * 'b) list) : 'b =
  match l with
   | [] -> raise Not_found
   | (k,v) :: t -> if c = k then v else assoc c t
```

```
# let v = [("p1",true);("p2",true);("p3",false)];;
val v:(string*bool) list=[("p1",true);("p2",true);("p3",false)]

# List.assoc "p2" v;;
- : bool = true

# (List.assoc "p4" v);;
  Exception: Not_found.
```