

(2) Fonctions (suite) – Listes

Programmation fonctionnelle (LU2IN019)

Licence d'informatique
2022/2023

Mathieu Jaume – Adrien Koutsos



Paramètres d'une fonction

- **let** $f (x : t_x) : t_r = e_f$
 - ▶ f est une fonction d'arité 1 (un seul paramètre : x)
- fonction à $n > 1$ paramètres
 - ▶ *exemple* : addition de deux entiers

let add (x : **int**) (y : **int**) : **int** = x + y

val add : $\underbrace{\text{int}}_{\substack{\text{type du} \\ \text{paramètre 1} \\ x}} \rightarrow \underbrace{\text{int}}_{\substack{\text{type du} \\ \text{paramètre 2} \\ y}} \rightarrow \underbrace{\text{int}}_{\substack{\text{type du} \\ \text{résultat} \\ x + y}} = \langle \text{fun} \rangle$

$\underbrace{\hspace{15em}}_{\substack{\text{fonction d'arité 2 : fonction à 1 paramètre} \\ \text{qui retourne une fonction d'arité 1}}}$

- ★ parenthésage implicite : **int** -> **int** -> **int** correspond au type **int** -> (**int** -> **int**)
- ★ il est possible d'appeler la fonction `add` avec un seul argument : **application partielle** (cf. cours 3)

Paramètres d'une fonction

• `let add (x : int) (y : int) : int = x + y`
`val add :`

type du paramètre 1
 x

type du paramètre 2
 y

type du résultat
 x + y

fonction d'arité 1

fonction d'arité 2 : fonction à 1 paramètre qui retourne une fonction d'arité 1

- fonction à $n > 1$ paramètres
 - solution (habituelle) : fonction unaire qui retourne une fonction d'arité $n - 1$

$\text{let } f(x_1 : t_1) \cdots (x_n : t_n) : t_r = e_f$

 - type de $f : t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow t_r$
c'est le type $t_1 \rightarrow (t_2 \rightarrow (\cdots \rightarrow (t_n \rightarrow t_r)))$
 - autre solution possible : le paramètre est un n -uplet
 - cf. cours 3 sur les types produits

Application (totale) d'une fonction

● **let** add (x : **int**) (y : **int**) : **int** = x + y

(add 2 3);;

- : **int** = 5

► (add 2 3) est interprété par ((add 2) 3)

► *exemples* **let** op (x : **int**) : **int** = -x

★ (add 2 op 3) et (add op 2 3) sont mal typés

★ (add 2 (op 3)) s'évalue à $2+(-3)=-1$

★ (add (op 2) 3) s'évalue à $(-2)+3=1$

★ (add add 2 3 4) et (add 2 add 3 4) sont mal typés

★ (add (add 2 3) 4) s'évalue à $(2+3)+4=5+4=9$

★ (add 2 (add 3 4)) s'évalue à $2+(3+4)=2+7=9$

● fonction à $n > 1$ paramètres

let f ($x_1 : t_1$) \cdots ($x_n : t_n$) : t_r = e_f

► application $(f\ e_1 \cdots e_n)$

★ chaque e_i est une expression de type t_i

★ l'expression $(f\ e_1 \cdots e_n)$ est de type t_r

★ $(f\ e_1 \cdots e_n)$ est interprété par $(((((f\ e_1)\ e_2) \cdots e_n))$

- structure linéaire dynamique
- type OCaml `'a list` prédéfini :
 - ▶ type somme : défini à partir de **constructeurs**
 - ★ constructeurs d'un type somme : seuls moyens pour construire des valeurs du type somme
 - ▶ type récursif : une liste non vide est définie par son premier élément et par la liste des éléments suivants
 - ▶ type polymorphe : paramétré par une variable de type (`'a`) désignant le type des éléments qui composent la liste
 - ★ une (valeur de type) `'a list` est une liste composée d'éléments de type `'a`
 - ★ structure homogène : toutes les valeurs contenues dans une liste ont le même type

Constructeurs de liste

- deux constructeurs polymorphes

<code>[]</code>	<code>'a list</code>	liste vide
<code>::</code>	<code>'a * 'a list -> 'a list</code>	liste non vide (constructeur utilisé en notation infixe)

- si

- ▶ `h` est une expression de type `'a`
- ▶ `t` est une `'a list`

alors `h :: t` est une `'a list` dont

- ▶ le premier élément est (le résultat de l'évaluation de) `h`
- ▶ la liste des éléments qui suivent le premier élément est (le résultat de l'évaluation de) la liste `t`

- exemple* : liste d'entiers (de type `int list`) composée de l'entier 1, suivi de l'entier 2, suivi de l'entier 3

```
# 1 :: (2 :: (3 :: []));;
- : int list = [1; 2; 3]
# (0 + 1) :: ((1 * 2) :: ((2 + 1) :: []));;
- : int list = [1; 2; 3]
```

Constructeurs de liste (syntaxe)

- deux constructeurs polymorphes

<code>[]</code>	<code>'a list</code>	liste vide
<code>::</code>	<code>'a * 'a list -> 'a list</code>	liste non vide

- syntaxe abrégée pour les listes

$[e_1; e_2; \dots; e_n] = e_1 :: (e_2 :: \dots (e_n :: []) \dots)$

- exemples*

```
# [];;  
- : 'a list = []  
  
# [18];;  
- : int list = [18]  
  
# [1; 2; 3];;  
- : int list = [1; 2; 3]  
  
# [1; (1 + 1); (2 + 1)];;  
- : int list = [1; 2; 3]  
  
# [5.2; (6.0 -. 5.4); 3.7];;  
- : float list = [5.2; 0.6; 3.7]  
  
# [0.4; 3; 5];;
```

This expression has type `int list` but is here used with type `float list`

Constructeurs de liste (syntaxe)

- syntaxe abrégée pour les listes

$[e_1; e_2; \dots; e_n] = e_1 :: (e_2 :: \dots (e_n :: []) \dots) = e_1 :: e_2 :: \dots :: e_n :: []$

- ▶ $::$ est associatif à droite : $e_1 :: e_2 :: e_3 = e_1 :: (e_2 :: e_3)$

★ $e_1 :: e_2 :: e_3$ de type `'a list`

$\rightsquigarrow e_1$ de type `'a` | $\rightsquigarrow e_2 :: e_3$ de type `'a list`

$\rightsquigarrow e_2$ de type `'a` $\rightsquigarrow e_3$ de type `'a list`

- type polymorphe `'a list`

- ▶ `'a` désigne un type quelconque

- ▶ liste de listes d'entiers : `'a` est le type `int list`

`[[1; 2]; []; [3; 4; 5]];`

- : `int list list = [[1; 2]; []; [3; 4; 5]]`

- exemples

`[] :: [];`

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

`1 :: [] :: [];`

Error: This expression has type `'a list` but an expression was expected of type `int`

`((1 :: []) :: []);;`

- : `int list list = [[1]]`

Filtrage (pattern matching)

- une liste est
 - ▶ soit la liste vide `[]`
 - ▶ soit une liste non vide `h :: t` construite à partir d'un premier élément `h` et d'une liste `t`
- expression de filtrage : `match ... with ...`

```
# match [1; 2; 3] with
| [] -> true
| h :: t -> false;;
```

```
- : bool = false
```

```
# match [] with
| [] -> true
| h :: t -> false;;
```

```
- : bool = true
```

```
# let x = [1; 2; 3] in
  match x with
  | [] -> true
  | h :: t -> false;;
```

```
- : bool = false
```

```
# let x = [] in
  match x with
  | [] -> true
  | h :: t -> false;;
```

```
- : bool = true
```

Filtrage (pattern matching)

- expression de filtrage

```
match x with  
| [] -> e1  
| h :: t -> e2
```

- ▶ [] et $h :: t$ sont des motifs de filtrage
 - ★ h et t sont des variables de filtrage
- ▶ e_1 et e_2 sont des expressions qui ont le même type
 - ★ elles définissent la valeur de l'expression de filtrage

- *exemple*

```
let is_empty (l : 'a list) : bool =  
  match l with  
  | [] -> true  
  | h :: t -> false
```

- ▶ is_empty est une fonction **polymorphe**
 - ★ elle peut s'appliquer sur des listes dont les éléments sont de type quelconque ('a)
- ```
(is_empty [1; 2; 3]);;
- : bool = false
(is_empty [true; true; false]);;
- : bool = false
```

# Filtrage (pattern matching)

- expression de filtrage

```
match x with
| [] -> e1
| h :: t -> e2
```

- ▶  $e_2$  peut utiliser les variables de filtrage  $h$  et  $t$

- *exemple* : somme des éléments d'une liste d'entiers  
fonction récursive sur une liste

- ▶  $(\text{sum\_l } []) = 0$
- ▶  $(\text{sum\_l } [e_1; e_2; \dots; e_n]) = e_1 + (\text{sum\_l } [e_2; \dots; e_n])$

```
let rec sum_l (l : int list) : int =
 match l with
 | [] -> 0
 | h :: t -> h + (sum_l t)
```

$$\begin{aligned}(\text{sum\_l } [4; 1; 3]) &= 4 + (\text{sum\_l } [1; 3]) \\ &= 4 + 1 + (\text{sum\_l } [3]) \\ &= 4 + 1 + 3 + (\text{sum\_l } []) = 4 + 1 + 3 + 0 = 8\end{aligned}$$

# Exhaustivité du filtrage

---

- *qui est exhaustif* : qui traite un sujet dans sa totalité, de manière complète
- définition d'une fonction `head` qui retourne le premier élément d'une liste
  - ▶ fonction de type `'a list -> 'a`
  - ▶ fonction non définie pour la liste vide

```
let head (l : 'a list) : 'a =
 match l with
 | h :: t -> h
```

Warning 8 [partial-match]: this pattern-matching is not exhaustive. Here is an example of a case that is not matched: []

```
(head [1; 2; 3]);;
- : int = 1
(head []);;
```

Exception: Match\_failure ("//toplevel//", 1, 28).

- définition d'une fonction `head` qui retourne le premier élément d'une liste
  - ▶ fonction de type `'a list -> 'a`
  - ▶ fonction non définie pour la liste vide
  - ▶ si l'argument de `head` est la liste vide :
    - ★ levée de l'exception prédéfinie `Invalid_argument`

```
let head (l : 'a list) : 'a =
 match l with
 | [] -> raise (Invalid_argument "empty list")
 | h :: t -> h
(head []);;
```

**Exception:** `Invalid_argument "empty list".`

↪ levée d'une exception : `raise`

## Variable(s) de filtrage « muette(s) »

---

- définition d'une fonction `head` qui retourne le premier élément d'une liste

```
let head (l : 'a list) : 'a =
 match l with
 | [] -> raise (Invalid_argument "empty list")
 | h :: t -> h
```

- ▶ l'expression `h` (à droite de `->`) ne dépend pas de la variable de filtrage (à gauche de `->`) `t`

- ★ `t` est une variable de filtrage « muette »
- ★ on peut remplacer `t` par le caractère `_`

```
let head (l : 'a list) : 'a =
 match l with
 | [] -> raise (Invalid_argument "empty list")
 | h :: _ -> h
```



- les motifs de filtrage peuvent contenir des constantes

```
let foo (l : 'a list) : bool =
 match l with
 | 2 :: _ -> true
 | _ -> false
(foo []);;
- : bool = false
(foo [1; 2; 3]);;
- : bool = false
(foo [2; 3; 4]);;
- : bool = true
```

- ordre des motifs de filtrage / filtrage redondant

```
let foo (l : 'a list) : bool =
 match l with
 | _ -> false
 | 2 :: _ -> true
(foo []);;
- : bool = false
(foo [1; 2; 3]);;
- : bool = false
(foo [2; 3; 4]);;
- : bool = false
```

Warning 11 [redundant-case]: this match case is unused.

- une variable de filtrage ne peut pas avoir plusieurs occurrences dans un motif de filtrage

```
let foo (l : 'a list) : bool =
 match l with
 | x :: (x :: _) -> true
 | _ -> false
```

Error: Variable x is bound several times in this matching

- expression du lien entre les variables de filtrage dans l'expression

```
let foo (l : 'a list) : bool =
 match l with
 | x :: (y :: _) -> x = y
 | _ -> false
```

# (foo [1; 2; 3]);;  
- : bool = false

# (foo [1; 1; 3]);;  
- : bool = true

# Ajout d'un élément dans une liste

- ajout d'un élément en tête de liste

- ▶ constructeur de liste `::` de type `'a -> 'a list -> 'a list`
- ▶ `::` permet d'ajouter un élément en tête d'une liste

- ajout d'un élément en fin de liste

```
let rec add_end (x : 'a) (l : 'a list) : 'a list =
 match l with
 | [] -> [x]
 | h :: t -> h :: (add_end x t)
```

```
(add_end 4 [1;2;3]) = 1 :: (add_end 4 [2;3]) = 1 :: (2 :: (add_end 4 [3]))
 = 1 :: (2 :: (3 :: (add_end 4 [])))
 = 1 :: (2 :: (3 :: [4])) = [1;2;3;4]
```

- ▶ le « coût » de l'ajout en fin de liste est proportionnel à la longueur de la liste

# Concaténation de deux listes

- *exemple* : la concaténation des deux listes `[6;3]` et `[4;1;3]` permet d'obtenir la liste `[6;3;4;1;3]`
  - fonction de type : `'a list -> 'a list -> 'a list`
  - le constructeur de liste `::`
    - ▶ de type `'a -> 'a list -> 'a list`
    - ▶ n'est pas un opérateur de concaténation
  - définition récursive de la concaténation de deux listes
    - ▶ `(app [] l) = l`
    - ▶ `(app [e1; e2; ...; en] l) = e1 :: (app [e2; ...; en] l)`
- ```
let rec app (l1 : 'a list) (l2 : 'a list) : 'a list =  
  match l1 with  
  | [] -> l2  
  | h :: t -> h :: (app t l2)
```

Concaténation de deux listes

- concaténation de deux listes

```
let rec app (l1 : 'a list) (l2 : 'a list) : 'a list =  
  match l1 with  
  | [] -> l2  
  | h :: t -> h :: (app t l2)
```

$(\text{app } [6; 3] \ [4; 1; 3]) = 6 :: (\text{app } [3] \ [4; 1; 3])$
 $= 6 :: (3 :: (\text{app } [] \ [4; 1; 3])) = 6 :: (3 :: [4; 1; 3]) = [6; 3; 4; 1; 3]$

- fonction OCaml prédéfinie de concaténation : @ (notation infixe)

```
# [1; 2; 3] @ [4; 4];;  
- : int list = [1; 2; 3; 4; 4]
```

- ajout d'un élément en fin de liste

```
let add_end (x : 'a) (l : 'a list) : 'a list = l @ [x]
```

- ▶ parcours de tous les éléments de la liste pour ajouter en fin de liste
- ▶ le « coût » de l'ajout en fin de liste est proportionnel à la longueur de la liste

Module List

module de la bibliothèque standard

- longueur d'une liste `List.length` : 'a list -> int

- ▶ définition récursive

- ★ $(\text{length } []) = 0$

- ★ $(\text{length } [e_1; e_2; \dots; e_n]) = 1 + (\text{length } [e_2; \dots; e_n])$

```
let rec length (l : 'a list) : int =  
  match l with  
  | [] -> 0  
  | _ :: t -> 1 + (length t)
```

$$\begin{aligned}(\text{length } [4; 1; 3]) &= 1 + (\text{length } [1; 3]) \\ &= 1 + 1 + (\text{length } [3]) \\ &= 1 + 1 + 1 + (\text{length } []) = 1 + 1 + 1 = 3\end{aligned}$$

- appartenance d'un élément à une liste

List.mem : 'a -> 'a list -> bool

- ▶ définition récursive

★ (mem e []) = false

★ (mem e [e₁; e₂; ...; e_n]) = true si e = e₁

★ (mem e [e₁; e₂; ...; e_n]) = (mem e [e₂; ...; e_n]) si e ≠ e₁

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

```
# (mem 3 [1; 2; 3]);;
```

```
- : bool = true
```

```
# (mem 3 [1; 2]);;
```

```
- : bool = false
```


- n -ième élément d'une liste

`List.nth` : 'a list \rightarrow int \rightarrow 'a

- définition récursive

★ $(\text{nth } [e_1; e_2; \dots; e_k] \ 0) = e_1$

★ $(\text{nth } [e_1; e_2; \dots; e_k] \ n) = (\text{nth } [e_2; \dots; e_k] \ (n - 1))$

```
let rec nth (l : 'a list) (n : int) : 'a =  
  if (n < 0) then raise (Invalid_argument "nth")  
  else  
    match l with  
    | [] -> raise (Failure "nth")  
    | h :: t -> if n = 0 then h else (nth t (n-1))
```

```
# (nth [1; 2; 3] 0);;
```

```
- : int = 1
```

```
# (nth [1; 2; 3] 2);;
```

```
- : int = 3
```

```
# (nth [1; 2; 3] (-5));;
```

```
Exception: Invalid_argument "nth".
```

```
# (nth [1; 2; 3] 8);;
```

```
Exception: Failure "nth".
```

- n -ième élément d'une liste

```
let rec nth (l : 'a list) (n : int) : 'a =  
  if (n < 0) then raise (Invalid_argument "nth")  
  else  
    match l with  
    | [] -> raise (Failure "nth")  
    | h :: t -> if n = 0 then h else (nth t (n-1))
```

- ▶ le test $n < 0$ est effectué à chaque appel récursif

- pour effectuer une unique fois le test $n < 0$:

```
let rec nth_aux (l : 'a list) (n : int) : 'a =  
  match l with  
  | [] -> raise (Failure "nth")  
  | h :: t -> if n = 0 then h else (nth_aux t (n-1))
```

```
let rec nth (l : 'a list) (n : int) : 'a =  
  if (n < 0) then raise (Invalid_argument "nth")  
  else nth_aux l n
```

- ▶ cf. cours 3 (fonction `nth_aux` locale à la fonction `nth`)

- inversion de l'ordre des éléments d'une liste

`List.rev` : 'a list \rightarrow 'a list

$(\text{rev } [e_1; e_2; \dots; e_n]) = [e_n; \dots; e_2; e_1]$

- définition récursive

★ $(\text{rev } []) = []$

★ $(\text{rev } [e_1; e_2; \dots; e_n]) = (\text{rev } [e_2; \dots; e_n]) @ [e_1]$

```
let rec rev (l : 'a list) : 'a list =  
  match l with  
  | [] -> []  
  | h :: t -> (rev t) @ [h]
```

$$\begin{aligned}(\text{rev } [1; 2; 3]) &= (\text{rev } [2; 3]) @ [1] = ((\text{rev } [3]) @ [2]) @ [1] \\ &= (((\text{rev } []) @ [3]) @ [2]) @ [1] \\ &= ([3] @ [2]) @ [1] = \boxed{\dots} = [3; 2; 1]\end{aligned}$$

– coût du calcul de $\boxed{\dots}$ de l'ordre de N^2 (où N est la longueur de la liste) : **c'est trop !**

Module List

- **Solution** : utiliser une liste temporaire `acc` pour stocker la liste partiellement inversée

- ▶ `acc` est appelé un accumulateur
- ▶ nécessaire d'utiliser une fonction auxiliaire `rev_aux`

```
let rec rev_aux (l : 'a list) (acc : 'a list) : 'a list =  
  match l with  
  | [] -> acc  
  | h :: t -> rev_aux t (h :: acc)
```

```
let rev (l : 'a list) : 'a list = rev_aux l []  
  
rev [1;2;3] = rev_aux [1;2;3] []  
            = rev_aux [2;3] [1]  
            = rev_aux [3] [2;1]  
            = rev_aux [] [3;2;1] = [3;2;1]
```

- ▶ temps d'exécution linéaire dans la longueur de la liste
- ▶ cf. cours 3 (fonction `rev_aux` locale à la fonction `rev`)
- ▶ cf. cours 6 (récursivité terminale)