Cours 3 : Listes, itérateurs de listes et tris

2019 - 2020

Un type récursif déjà vu : les entiers

Un type récursif déjà vu : les entiers

Types récursifs pour les entiers

- Les entiers naturels (donc positifs) correspondent à plusieurs schémas récursifs et peuvent être vus comme un type récursif (de différentes façons).
- Un **entier** est de la forme :
  - soit `0` *(* cas de base/terminal *)*
  - soit `n+1` où `n` est un **entier**. *(* cas général/récursif *)*
- Ou bien, un **entier** est de la forme :
  - soit `0` *(* cas de base/terminal *)*
  - soit `2*n+2` où `n` est un **entier**. *(* cas général/récursif *)*
  - soit `2*n+1` où `n` est un **entier**. *(* cas général/récursif *)*
- Ce sont des définitions d'un type récursif.
- D'une définition d'un type récursif découle naturellement des fonctions récursives (même récursion que le type), par exemple la factorielle pour le premier schéma, la puissance indienne pour le second.

Structure de données: liste

Structure de données: liste

Définition du type des listes ♪♪♪

Définition : une `'a list` est :

- soit la liste vide *(* cas de base/terminal *)*
  notée `[]` et de type `'a list` .
- soit `a::l` *(* cas général/récursif *)*
  où `a` est **un** élément de type `'a` et `l` une `'a list` .

Remarques

- la structure de données “liste” est **homogène**, i.e. tous les éléments d’une liste ont le même type α. ♪♪♪
- une liste non vide se présente toujours sous la forme `tete::queue`, les différents éléments ne sont accessibles que de cette façon.
  $\Rightarrow$ Pas d’accès direct indexé comme pour les tableaux. ♪♪♪
- structure de données **dynamique**: on peut “ajouter” ou “retirer” des éléments (ce n’est qu’un abus de langage, puisqu’il n’y a pas d’effets de bord). ♪♪♪

Structure de données: liste

Exemples

```
# [];;                    (* la liste vide *)
- : 'a list = []
# 1::2::3::[];;
- : int list = [1;2;3]
# [[1;2];[];[3]];;
- : int list list = [[1;2];[];[3]]
```

On remarque l'équivalence des écritures

a::b::c ::[] et [a; b; c].

Structure de données: liste

Accès à la tête et à la queue d'une liste ♪♪♪ puis ♪♪♪

- On accède à la tête (resp. queue) d'une liste à l'aide de la fonctions
  List.hd (head) (resp. List.tl (tail))
  ou en utilisant le filtrage :

```
let rec somme_liste liste =
  match liste with
  | []              -> 0                              (* cas de base/terminal *)
  | tete::queue -> tete + (somme_liste queue) (* cas general/recursif *)
```

- Cette fonction suit la récursivité naturelle (structurelle) des listes.

```
let rec firsts liste =
  match liste with
  | []              -> 0                          (* cas de base/terminal *)
  | (f,_)::queue ->f::(firsts queue) (* cas general/recursif *)
```

```
# let (f,_)::queue = [(1,2);(3,4);(5,6)];;
val f : int = 1
val queue : (int * int) list = [(3, 4); (5, 6)]
```

Structure de données: liste

Exercices

1. Écrire les fonctions `hd` et `tl`. ♪♪♪
2. Écrire la fonction `taille` qui renvoie la longueur d'une liste. ♪♪♪
3. Écrire la fonction `append` qui renvoie la concaténation de deux listes. Quelle est sa complexité? ♪♪♪

Itérateurs de listes

Itérateurs de liste

♪♪♪

List.map

```
List.map f [t1;t2 ... ;tn] = [f t1;f t2 ... ;f tn]
```

Exemple

```
List.map (fun e -> e+1) [1;2;3;4] = [2;3;4;5]
List.map (fun (a,_) -> a) [(1,'a');(2,'b');(3,'c')] = [1;2;3]
List.map List.fst [(1,'a');(2,'b');(3,'c')] = [1;2;3]
```

Itérateurs de liste

Exercice

1. Donner le type de l'itérateur `List.map`

2. Écrire cet itérateur.

3. Écrire `string_of_int_list`, qui transforme une liste d'entiers en une liste de chaînes de caractères, en utilisant `List.map`.

Itérateurs de liste

Exercice

1. Donner le type de l'itérateur `List.map`

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

2. Écrire cet itérateur.

3. Écrire `string_of_int_list`, qui transforme une liste d'entiers en une liste de chaînes de caractères, en utilisant `List.map`.

Itérateurs de liste

Exercice

1. Donner le type de l'itérateur List.map

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

2. Écrire cet itérateur.

```
let rec map f l =
  match l with
   |[]  -> []
   | t :: q -> (f t )::( map f q)
```

3. Écrire string_of_int_list , qui transforme une liste d'entiers en une liste de chaînes de caractères, en utilisant List.map.

Itérateurs de liste

Exercice

1. Donner le type de l'itérateur List.map

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

2. Écrire cet itérateur.

```
let rec map f l =
  match l with
   |[]  -> []
   | t :: q -> (f t )::( map f q)
```

3. Écrire string_of_int_list , qui transforme une liste d'entiers en une liste de chaînes de caractères, en utilisant List.map.

```
let string_of_int_list l = List.map (fun e -> string_of_int e) l
```

Itérateurs de liste

Exercice

1. Donner le type de l'itérateur List.map

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

2. Écrire cet itérateur.

```
let rec map f l =
  match l with
   |[]  -> []
   | t :: q -> (f t )::( map f q)
```

3. Écrire string_of_int_list , qui transforme une liste d'entiers en une liste de chaînes de caractères, en utilisant List.map.

```
let string_of_int_list l = List.map (fun e -> string_of_int e) l
```

```
let string_of_int_list l = List.map string_of_int l
```

Itérateurs de liste

Exercice

1. Donner le type de l'itérateur `List.map`

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

2. Écrire cet itérateur.

```
let rec map f l =
  match l with
   |[]  -> []
   | t :: q -> (f t )::( map f q)
```

3. Écrire `string_of_int_list`, qui transforme une liste d'entiers en une liste de chaînes de caractères, en utilisant `List.map`.

```
let string_of_int_list l = List.map (fun e -> string_of_int e) l
```

```
let string_of_int_list l = List.map string_of_int l
```

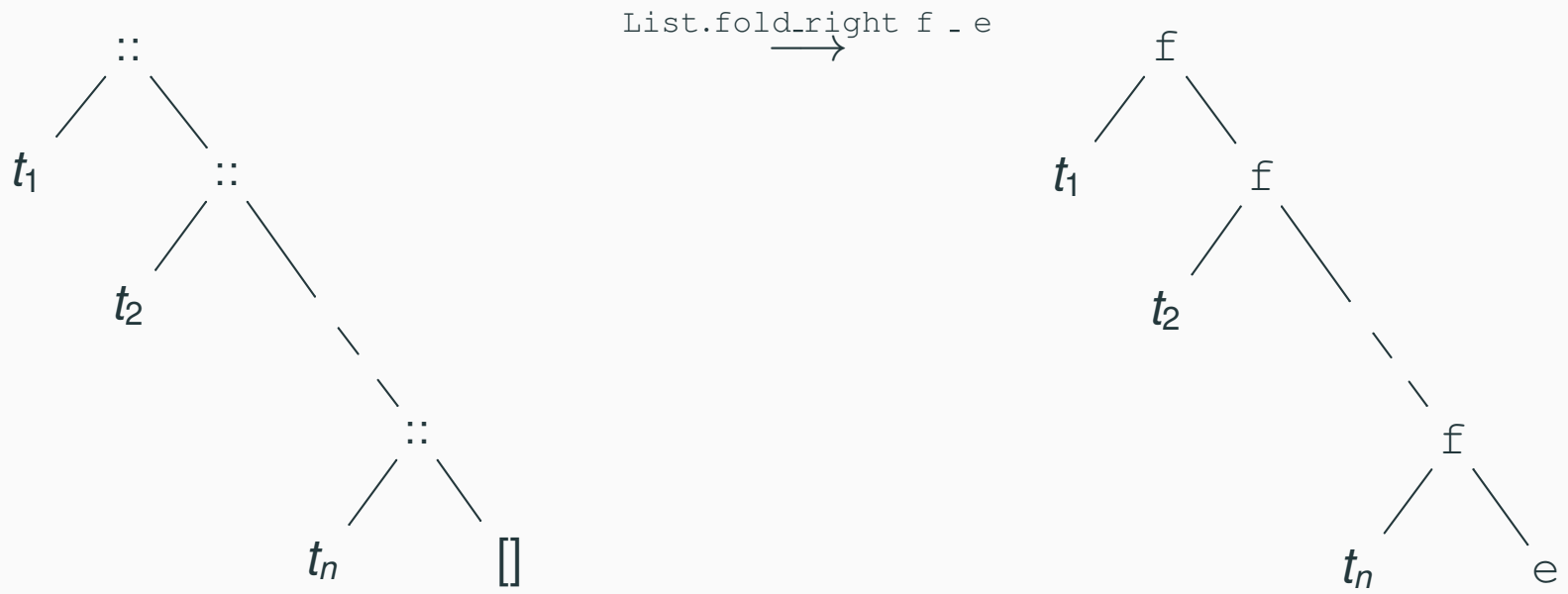
```
let string_of_int_list = List.map string_of_int
```

Itérateurs de liste

♪♪♪

List . fold_right

List . fold_right f [t_1;t_2; . . . ;t_n] e = (f t_1 (f t_2 (. . . (f t_n e) . . .)))

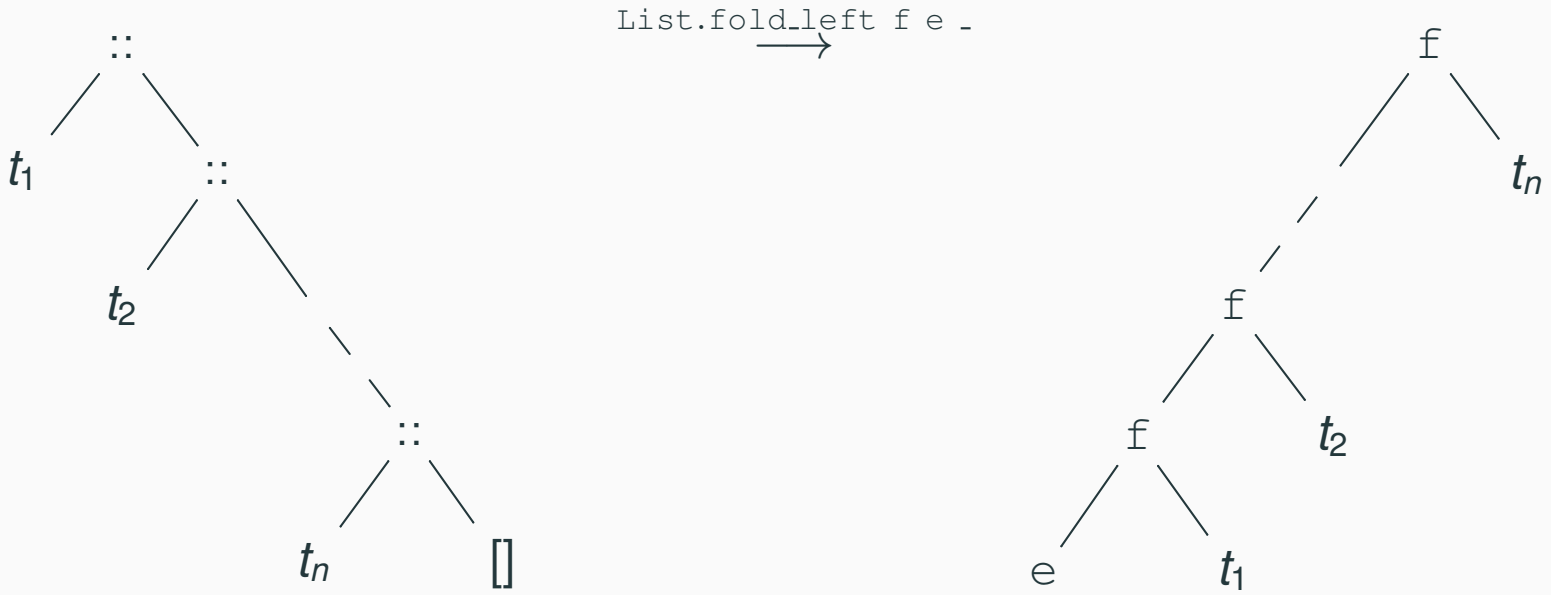


Itérateurs de liste

♪♪♪

List . fold_left

List . fold_left f e [t_1;t_2; . . . ;t_n] = (f (. . .(f (f e t_1) t_2) . . .) t_n)



Itérateurs de liste

Exercice

1. Donner le type des itérateurs `List.fold_right` et `List.fold_left`. ♪♪♪

2. Écrire ces itérateurs. (cf vidéos de chaque itérateur)

3. Écrire la fonction `rev` (qui renverse une liste) à l'aide des deux itérateurs. Quelle version a la complexité la plus faible? (cf vidéos du `List.fold_left`)

Itérateurs de liste

Exercice

1. Donner le type des itérateurs `List.fold_right` et `List.fold_left`. ♪♪♪

```
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

2. Écrire ces itérateurs. (cf vidéos de chaque itérateur)

3. Écrire la fonction `rev` (qui renverse une liste) à l'aide des deux itérateurs. Quelle version a la complexité la plus faible? (cf vidéos du `List.fold_left`)

Itérateurs de liste

Exercice

1. Donner le type des itérateurs `List.fold_right` et `List.fold_left`. ♪♪♪

```
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

2. Écrire ces itérateurs. (cf vidéos de chaque itérateur)

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

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

3. Écrire la fonction `rev` (qui renverse une liste) à l'aide des deux itérateurs. Quelle version a la complexité la plus faible? (cf vidéos du `List.fold_left`)

Itérateurs de liste

Exercice

1. Donner le type des itérateurs List.fold_right et List.fold_left. ♪♪♪

```
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

2. Écrire ces itérateurs. (cf vidéos de chaque itérateur)

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

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

3. Écrire la fonction rev (qui renverse une liste) à l'aide des deux itérateurs. Quelle version a la complexité la plus faible? (cf vidéos du List.fold_left)

```
let rev l = List.fold_left (fun accu t -> t::accu) [] l
let rev l = List.fold_right (fun t rev_q -> rev_q@[t]) l []
```

Itérateurs de liste

Autres itérateurs

`https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`

Les avantages des itérateurs structurels

- garantit la terminaison
- simplifie les calculs de complexité
- permet la simplification, l'optimisation de code

Tris ♪♪♪

Complexité des tris

La meilleure complexité (en terme de comparaison d'éléments deux à deux) possible pour un algorithme de tri par comparaison est en $O(n * log\ n)$.

Une implémentation immédiate

- Calculer toutes les permutations et ne garder que celle qui est bien ordonnée.
- Complexité : $\Theta((n-1) * n!)$
  - $n!$ permutations
  - $n-1$ comparaisons d'éléments par permutation

Tri par insertion - Analyse récursive

- Si je sais trier une liste de taille $n-1$, comment puis-je trier une liste de taille n?
- J'insère l'élément souhaité à "sa place".

$\Rightarrow$ Besoin d'une fonction auxiliaire qui insère un élément dans une liste triée.

Tri par insertion - Code

```
(* insertion : 'a -> 'a list -> 'a list *)
(* insere un elt dans une liste triee par ordre croissant *)
let rec insertion x liste =
 match liste with
 | []              -> [x]
 | tete::queue -> if x <= tete then x::liste
                                  else tete::( insertion x queue)

(* tri_insertion : 'a list -> 'a list *)
(* trie une liste par ordre croissant *)
let rec tri_insertion liste =
 match liste with
 | []              -> []
 | tete::queue -> insertion tete ( tri_insertion queue)
```

OU BIEN

```
let tri_insertion l = List.fold_right insertion l []
```

Tri par insertion - Complexité

- On étudie la complexité du pire cas, correspondant à l'insertion en fin de liste.

$$\begin{aligned} C_{max}(0) &= 0 \\ C_{max}(n+1) &= n + C_{max}(n) \end{aligned}$$

- $C_{max}(n)$ est la somme des $n-1$ premiers entiers

$$C_{max}(n) = \frac{n(n-1)}{2}$$

- D'où: $C_{max}(n) = \Theta(n^2)$
- $C_{min}(n) = \Theta(n)$: tri d'une liste déjà triée, l'insertion se fait en une unique comparaison

Tri fusion - Analyse récursive

- Si je sais trier une liste de taille $n/2$ comment puis-je trier une liste de taille n?
- Réponse : Je fusionne deux listes triées.

$\Rightarrow$ Besoin de deux fonctions auxiliaires :

- une qui découpe une liste en deux sous-listes de même taille ± 1.
- une qui fusionne deux listes triées

Tri fusion - Code ♪♪♪

```
(* decompose : 'a list -> 'a list * 'a list *)
(* decompose une liste en deux listes de tailles egales (+/- un elt) *)
let rec decompose liste =
 match liste with
 | []              -> [], []
 | [_]             -> liste, []
 | e1::e2::queue -> let (l1,l2)= decompose queue in (e1::l1, e2::l2)

(* recompose : 'a list -> 'a list -> 'a list *)
(* fusionne deux listes triees par ordre croissant *)
(* pour en faire une seule triee par ordre croissant *)
let rec recompose liste1 liste2 =
 match liste1, liste2 with
 | []              , _             -> liste2
 | _               , []            -> liste1
 | tete1 ::queue1, tete2::queue2 -> if tete1 < tete2
                                    then tete1 :: recompose queue1 liste2
                                    else tete2 :: recompose liste1 queue2
```

Tri fusion - Code

L'algorithme de tri consiste alors à :

- couper la liste en deux
- trier les deux sous-listes (appels récursifs)
- fusionner les deux sous-listes triées

```
(* tri_fusion : 'a list -> 'a list *)
(* trie une liste par ordre croissant *)
let rec tri_fusion liste =
 match liste with
 | [] -> []
 | [_] -> liste
 | _ -> let (l1,l2) = decompose liste in recompose (tri_fusion l1) ( tri_fusion l2)
```

Tri fusion - Complexité

- Le tri fusion est un algorithme de complexité uniforme, quelles que soient les données. Ainsi, $C_{min}(n) = C_{moy}(n) = C_{max}(n) = C(n)$
- $$\begin{aligned} C(0) &= 0 \\ C(1) &= 0 \\ C(2n+2) &= 2C(n+1) + 2n + 1 \\ C(2n+3) &= C(n+2) + C(n+1) + 2n + 2 \end{aligned}$$
- $$C(2^{\lceil \log_2 n \rceil - 1}) < C(n) \leq C(2^{\lceil \log_2 n \rceil})$$
- $$\begin{aligned} C(2^0) &= C(1) = 0 \\ C(2^{n+1}) &= 2^{n+1} - 1 + 2C(2^n) \end{aligned}$$
- On pose $D(n) = \frac{C(2^n)}{2^n}$
- $$\begin{aligned} D(0) &= \frac{C(1)}{1} = 0 \\ D(n+1) &= 1 - \frac{1}{2^{n+1}} + \frac{2C(2^n)}{2^{n+1}} = 1 - \frac{1}{2^{n+1}} + D(n) \end{aligned}$$

Tri fusion - Complexité

- Par intégration de $D(n)$:

$$\begin{aligned} D(n) &= \underbrace{(1-\frac{1}{2^n})+(1-\frac{1}{2^{n-1}})+\ldots+(1-\frac{1}{2^1})}_{n \text{ termes}} \\ &= n-\sum_{k=1}^{n}\frac{1}{2^k} \\ &= n-1+\frac{1}{2^n} \end{aligned}$$

- $C(2^n) = (n-1)2^n + 1$
-

$$1+(\lceil \log_2 n \rceil - 2) * 2^{\lceil \log_2 n \rceil - 1} < C(n) \leq 1+(\lceil \log_2 n \rceil - 1) * 2^{\lceil \log_2 n \rceil}$$

- Puisque $\lceil \log_2 n \rceil = \Theta(\log_2 n)$ et $2^{\lceil \log_2 n \rceil} = \Theta(2^{\log_2 n}) = \Theta(n)$, on obtient finalement $C(n) = \Theta(n * \log n)$