Listes

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

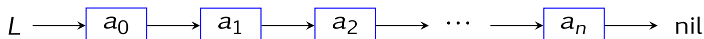
Structure de donnée

C'est la description d'une structure logique destinée à organiser et à agir sur des données indépendamment de la mise en œuvre effective de ces structures.

Structure de donnée

C'est la description d'une structure logique destinée à organiser et à agir sur des données indépendamment de la mise en œuvre effective de ces structures.

Une liste est une collection séquentielle et de taille arbitraire de données de même type : chaque élément possède, en plus de la donnée, d'un pointeur vers l'élément suivant de la liste.

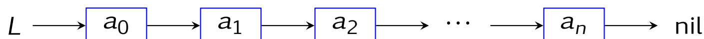


Le *nil* est un élément particulier caractérisant la terminaison de la liste.

Structure de donnée

C'est la description d'une structure logique destinée à organiser et à agir sur des données indépendamment de la mise en œuvre effective de ces structures.

Une liste est une collection séquentielle et de taille arbitraire de données de même type : chaque élément possède, en plus de la donnée, d'un pointeur vers l'élément suivant de la liste.



Le *nil* est un élément particulier caractérisant la terminaison de la liste.

Ainsi, le type de données abstrait définissant une liste est le suivant :

$$\text{Liste} = \text{nil} + \text{Élément} \times \text{Liste}$$

Mise en œuvre

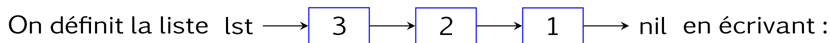
$$\text{Liste} = \text{nil} + \text{Élément} \times \text{Liste}$$

```
# type 'a liste = Nil | Cellule of 'a * ('a liste) ;;
Type liste defined.
```

Mise en œuvre

$$\text{Liste} = \text{nil} + \text{Élément} \times \text{Liste}$$

```
# type 'a liste = Nil | Cellule of 'a * ('a liste) ;;
Type liste defined.
```




```
# let lst = Cellule (3, Cellule (2, Cellule (1, Nil))) ;;
lst : int liste = Cellule (3, Cellule (2, Cellule (1, Nil)))
```

Mise en œuvre

$$\text{Liste} = \text{nil} + \text{Élément} \times \text{Liste}$$

```
# type 'a liste = Nil | Cellule of 'a * ('a liste) ;;
Type liste defined.
```

On crée un constructeur :

```
# let construire t q = Cellule (t, q) ;;
construire : 'a -> 'a liste -> 'a liste = <fun>
```

```
# let lst2 = construire 4 lst ;;
lst2 : int liste = Cellule (4, Cellule (3, Cellule (2,
                                          Cellule (1, Nil))))
```

Mise en œuvre

$$\text{Liste} = \text{nil} + \text{Élément} \times \text{Liste}$$

```
# type 'a liste = Nil | Cellule of 'a * ('a liste) ;;
Type liste defined.
```

À l'inverse, on récupère les données en définissant les fonctions :

```
# let tete = function
    | Nil            -> failwith "liste_vide"
    | Cellule (t, q) -> t ;;
tete : 'a liste -> 'a = <fun>
```

```
# let queue = function
    | Nil            -> failwith "liste_vide"
    | Cellule (t, q) -> q ;;
queue : 'a liste -> 'a liste = <fun>
```

Mise en œuvre

$$\text{Liste} = \text{nil} + \text{Élément} \times \text{Liste}$$

```
# type 'a liste = Nil | Cellule of 'a * ('a liste) ;;
Type liste defined.
```

En conclusion :

- les listes peuvent grossir dynamiquement, mais les éléments sont toujours rajoutés en tête de liste ;
- on ne peut accéder directement qu'à la tête de la liste.

Mise en œuvre

$$\text{Liste} = \text{nil} + \text{Élément} \times \text{Liste}$$

```
# type 'a liste = Nil | Cellule of 'a * ('a liste) ;;
Type liste defined.
```

En conclusion :

- les listes peuvent grossir dynamiquement, mais les éléments sont toujours rajoutés en tête de liste ;
- on ne peut accéder directement qu'à la tête de la liste.

Nous allons maintenant étudier le type *'a list* prédéfini en Caml.

Description d'une liste Caml

Les listes d'éléments de type *'a* ont pour type : *'a list* .

Les listes sont délimitées par des crochets [et], les éléments (qui doivent être de même type) sont séparés par un point-virgule.

```
# [4; 3; 2; 1] ;;
- : int list = [4; 3; 2; 1]
# [`a`; `b`; `c`] ;;
- : char list = [`a`; `b`; `c`]
```

```
# [4; `a`; 3; 2; 1] ;;
Entrée interactive:
>[4; `a`; 3; 2; 1] ;;
>^^^^^^^^^^^^^^^^^^
This expression has type int list, but is used with type char list.
```

L'élément nil sera représenté par la liste vide [] et aura pour type *'a list* .

Description d'une liste CAML

Les listes d'éléments de type *'a* ont pour type : *'a list* .

Insertion en tête de liste par l'opérateur infixe **::**

```
# 5::[4; 3; 2; 1] ;;
- : int list = [5; 4; 3; 2; 1]
# ‘a‘::‘b‘::‘c‘::[] ;;
- : char list = [‘a‘; ‘b‘; ‘c‘]
```

Les fonctions **hd** et **tl** permettent d'obtenir la tête et la queue d'une liste :

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

```
# hd [] ;;
Uncaught exception: Failure "hd"
```

Filtrage et récursivité

Le motif **t::q** est reconnu par toute liste non vide, et dans la suite de l'évaluation **t** prend la valeur de la tête et **q** celle de la queue.

La redéfinition de **hd** :

```
# let hd = function                        (* fonction prédéfinie en Caml *)
    | []   -> failwith "hd"
    | t::q -> t ;;
hd : 'a list -> 'a = <fun>
```

La redéfinition de **tl** :

```
# let tl = function                        (* fonction prédéfinie en Caml *)
    | []   -> failwith "tl"
    | t::q -> q ;;
tl : 'a list -> 'a list = <fun>
```

Filtrage et récursivité

Le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prend la valeur de la tête et **`q`** celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- `[x]`

Filtrage et récursivité

Le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prend la valeur de la tête et **`q`** celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- `[x]` → toute *'a list* à un élément.
- **`x::[]`**

Filtrage et récursivité

Le motif `t::q` est reconnu par toute liste non vide, et dans la suite de l'évaluation `t` prend la valeur de la tête et `q` celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- `[x]` → toute *'a list* à un élément.
- `x::[]` → toute *'a list* à un élément.
- `[]::x`

Filtrage et récursivité

Le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prend la valeur de la tête et **`q`** celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- **`[x]`** → toute *'a list* à un élément.
- **`x::[]`** → toute *'a list* à un élément.
- **`[]::x`** → toute *'a list list* dont le 1er élément est la liste vide.
- **`[1; 2; x]`**

Filtrage et récursivité

Le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prend la valeur de la tête et **`q`** celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- **`[x]`** → toute *'a list* à un élément.
- **`x::[]`** → toute *'a list* à un élément.
- **`[]::x`** → toute *'a list list* dont le 1er élément est la liste vide.
- **`[1; 2; x]`** → toute *int list* à trois éléments débutant par 1 et 2.
- **`1::2::[x]`**

Filtrage et récursivité

Le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prend la valeur de la tête et **`q`** celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- **`[x]`** → toute *'a list* à un élément.
- **`x::[]`** → toute *'a list* à un élément.
- **`[]::x`** → toute *'a list list* dont le 1er élément est la liste vide.
- **`[1; 2; x]`** → toute *int list* à trois éléments débutant par 1 et 2.
- **`1::2::[x]`** → toute *int list* à trois éléments débutant par 1 et 2.
- **`1::2::x`**

Filtrage et récursivité

Le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prend la valeur de la tête et **`q`** celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- **`[x]`** → toute *'a list* à un élément.
- **`x::[]`** → toute *'a list* à un élément.
- **`[]::x`** → toute *'a list list* dont le 1er élément est la liste vide.
- **`[1; 2; x]`** → toute *int list* à trois éléments débutant par 1 et 2.
- **`1::2::[x]`** → toute *int list* à trois éléments débutant par 1 et 2.
- **`1::2::x`** → toute *int list* débutant par 1 et 2.
- **`x::y::z`**

Filtrage et récursivité

Le motif `t::q` est reconnu par toute liste non vide, et dans la suite de l'évaluation `t` prend la valeur de la tête et `q` celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- `[x]` → toute *'a list* à un élément.
- `x::[]` → toute *'a list* à un élément.
- `[]::x` → toute *'a list list* dont le 1er élément est la liste vide.
- `[1; 2; x]` → toute *int list* à trois éléments débutant par 1 et 2.
- `1::2::[x]` → toute *int list* à trois éléments débutant par 1 et 2.
- `1::2::x` → toute *int list* débutant par 1 et 2.
- `x::y::z` → toute *'a list* de longueur $\geqslant 2$.

Filtrage et récursivité

Le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prend la valeur de la tête et **`q`** celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- `[x]` → toute *'a list* à un élément.
- `x::[]` → toute *'a list* à un élément.
- `[]::x` → toute *'a list list* dont le 1er élément est la liste vide.
- `[1; 2; x]` → toute *int list* à trois éléments débutant par 1 et 2.
- `1::2::[x]` → toute *int list* à trois éléments débutant par 1 et 2.
- `1::2::x` → toute *int list* débutant par 1 et 2.
- `x::y::z` → toute *'a list* de longueur $\geqslant 2$.
- `x::x::y`

Filtrage et récursivité

Le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prend la valeur de la tête et **`q`** celle de la queue.

Exercice. Décrire les listes reconnues par les motifs suivants :

- **`[x]`** → toute *'a list* à un élément.
- **`x::[]`** → toute *'a list* à un élément.
- **`[]::x`** → toute *'a list list* dont le 1er élément est la liste vide.
- **`[1; 2; x]`** → toute *int list* à trois éléments débutant par 1 et 2.
- **`1::2::[x]`** → toute *int list* à trois éléments débutant par 1 et 2.
- **`1::2::x`** → toute *int list* débutant par 1 et 2.
- **`x::y::z`** → toute *'a list* de longueur $\geqslant 2$.
- **`x::x::y`** → motif incorrect : The variable x is bound several times in this pattern.

Filtrage et récursivité

Le motif **t::q** est reconnu par toute liste non vide, et dans la suite de l'évaluation **t** prend la valeur de la tête et **q** celle de la queue.

- Calcul de la longueur d'une liste :

```
let rec list_length = function    (* prédéfinie en Caml *)
  | []   -> 0
  | t::q -> 1 + list_length q ;;
```

Cette fonction a une complexité en $\Theta(\ell)$ où ℓ est la longueur de la liste.

- Obtention du dernier élément d'une liste :

```
let rec last = function
  | []   -> failwith "last"
  | [a]  -> a
  | _::q -> last q ;;
```

Cette fonction a une complexité en $\Theta(\ell)$.

Filtrage et récursivité

Le motif **t::q** est reconnu par toute liste non vide, et dans la suite de l'évaluation **t** prend la valeur de la tête et **q** celle de la queue.

- Le test d'appartenance à une liste :

```
let rec mem x = function          (* prédéfinie en Caml *)
  | []              -> false
  | t::_ when t = x -> true
  | _::q            -> mem x q ;;
```

Cette fonction a une complexité en $O(\ell)$.

Notons que le principe de l'évaluation paresseuse permet d'écrire de manière équivalente :

```
let rec mem x = function
  | []    -> false
  | t::q  -> (t = x) || mem x q ;;
```

Filtrage et récursivité

Le motif **t::q** est reconnu par toute liste non vide, et dans la suite de l'évaluation **t** prend la valeur de la tête et **q** celle de la queue.

- L'obtention du n^e élément d'une liste :

```
let rec nth n = function
  | []                -> raise Not_found
  | t::_ when n = 1 -> t
  | _::q              -> nth (n - 1) q ;;
```

- La concaténation de deux listes :

```
let rec concat lst1 lst2 = match lst1 with
  | []   -> lst2
  | t::q -> t::(concat q lst2) ;;
```

L'instruction `@` est la forme infixe de cette fonction :

```
# [1; 2; 3] @ [4; 5; 6] ;;
- : int list = [1; 2; 3; 4; 5; 6]
```

Autrement dit, **let concat = prefix @** est équivalent.

Fonctionnelles agissant sur les listes

- La fonction **map**

Étant données une fonction f de type *'a -> 'b* et une liste $[a_0; \cdots; a_{n-1}]$ de type *'a list*, la fonctionnelle **map** crée la liste $[f(a_0); \cdots; f(a_{n-1})]$.

```
# let rec map f = function      (* prédéfinie en Caml *)
    | []   -> []
    | t::q -> (f t)::(map f q) ;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Par exemple :

```
# map string_length ["alpha"; "beta"; "gamma"; "delta"] ;;
- : int list = [5; 4; 5; 5]
```

Fonctionnelles agissant sur les listes

- La fonction **map**

Étant données une fonction f de type *'a -> 'b* et une liste $[a_0; \cdots; a_{n-1}]$ de type *'a list*, la fonctionnelle **map** crée la liste $[f(a_0); \cdots; f(a_{n-1})]$.

```
# let rec map f = function     (* prédéfinie en Caml *)
    | []    -> []
    | t::q -> (f t)::(map f q) ;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Par exemple :

```
# map string_length ["alpha"; "beta"; "gamma"; "delta"] ;;
- : int list = [5; 4; 5; 5]
```

Exercice. Déterminer le type et le rôle de :

- `let myst1 l = (map fst l), (map snd l) ;;`

Fonctionnelles agissant sur les listes

- La fonction **map**

Étant données une fonction f de type *'a -> 'b* et une liste $[a_0; \cdots; a_{n-1}]$ de type *'a list*, la fonctionnelle **map** crée la liste $[f(a_0); \cdots; f(a_{n-1})]$.

```
# let rec map f = function     (* prédéfinie en Caml *)
    | []    -> []
    | t::q -> (f t)::(map f q) ;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Par exemple :

```
# map string_length ["alpha"; "beta"; "gamma"; "delta"] ;;
- : int list = [5; 4; 5; 5]
```

Exercice. Déterminer le type et le rôle de :

- **`let myst1 l = (map fst l), (map snd l) ;;`**

→ divise une *('a * 'b) list* en un couple *'a list * 'b list*

Fonctionnelles agissant sur les listes

- La fonction **do_list**

Étant données une fonction f de type *'a -> unit* et une liste $[a_0; \cdots; a_{n-1}]$ de type *'a list*, la fonctionnelle **do_list** effectue la séquence

$$f(a_0)\,;f(a_1)\,;\cdots;f(a_{n-1})$$

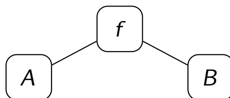
```
# let rec do_list (f : 'a -> unit) = function    (* prédéfinie *)
    | []   -> ()
    | t::q -> f t ; do_list f q ;;
do_list : ('a -> unit) -> 'a list -> unit = <fun>
```

Exemple :

```
do_list print_string ["alpha"; "beta"; "gamma"; "delta"] ;;
alphabetagammadelta- : unit = ()
```

Représentation arborescente d'une liste

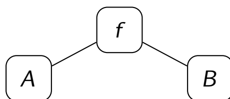
Si a et b sont des expressions, on convient de représenter l'expression $f(a, b)$ par l'arbre :



où A et B sont des arbres représentant les expressions a et b.

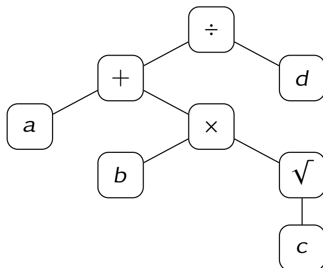
Représentation arborescente d'une liste

Si a et b sont des expressions, on convient de représenter l'expression $f(a,b)$ par l'arbre :



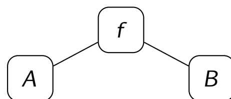
où A et B sont des arbres représentant les expressions a et b.

Exemple. L'expression $\dfrac{a+b\sqrt{c}}{d}$ est représentée par :



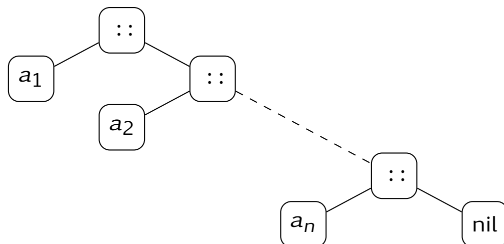
Représentation arborescente d'une liste

Si a et b sont des expressions, on convient de représenter l'expression $f(a,b)$ par l'arbre :



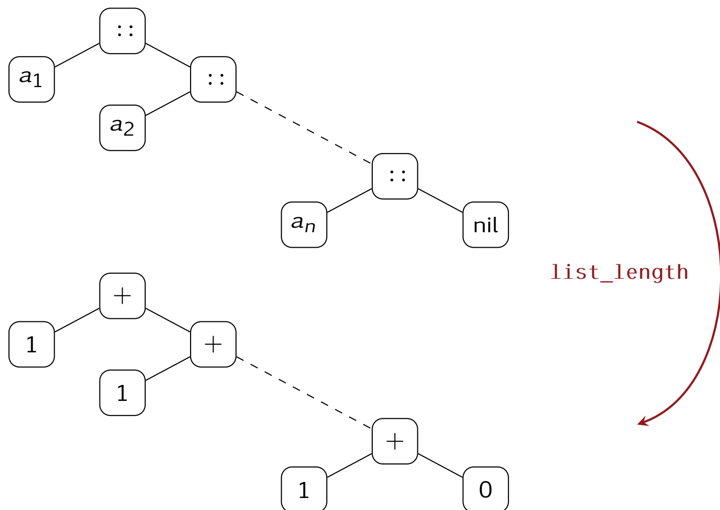
où A et B sont des arbres représentant les expressions a et b.

La liste $[a_1; a_2; \cdots; a_n]$ est représentée par :



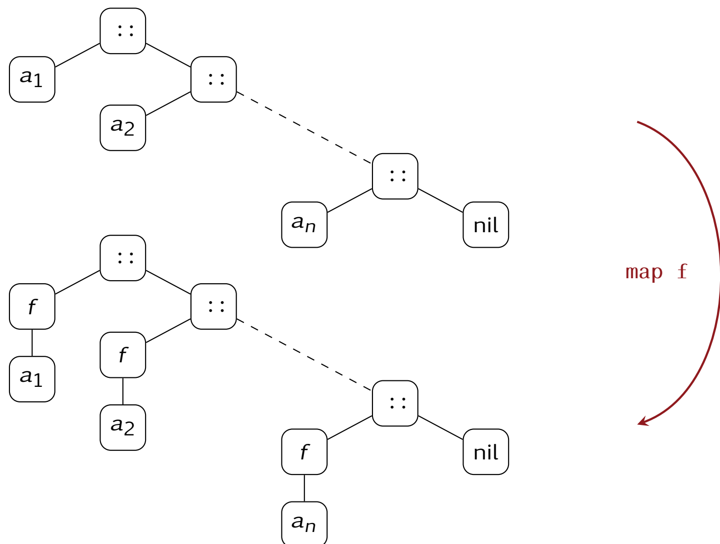
Réécriture d'une liste

Calculer la longueur d'une liste revient à effectuer la transformation :



Réécriture d'une liste

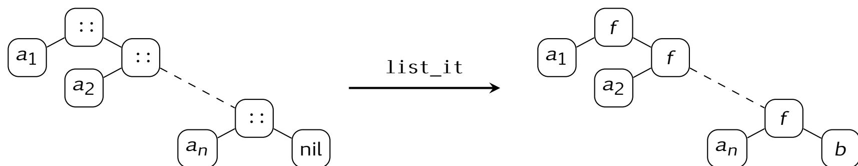
Appliquer la fonction **map f** revient à effectuer la transformation :



Fonctions d'ordre supérieur

La fonctionnelle list_it

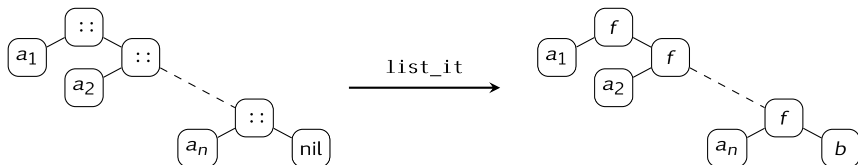
Elle a pour objet d'effectuer la transformation :



Fonctions d'ordre supérieur

La fonctionnelle list_it

Elle a pour objet d'effectuer la transformation :



On peut redéfinir la fonction **list_length** en posant :

```
let list_length lst = list_it (fun a b -> 1 + b) lst 0 ;;
```

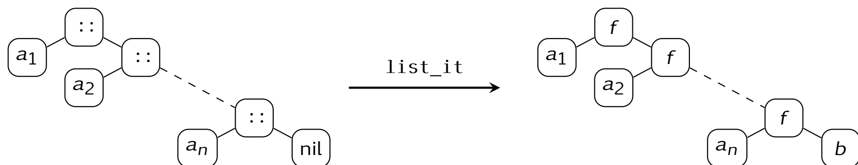
et redéfinir la fonction **map** en posant :

```
let map f lst = list_it (fun a b -> (f a)::b) lst [] ;;
```

Fonctions d'ordre supérieur

La fonctionnelle list_it

Elle a pour objet d'effectuer la transformation :



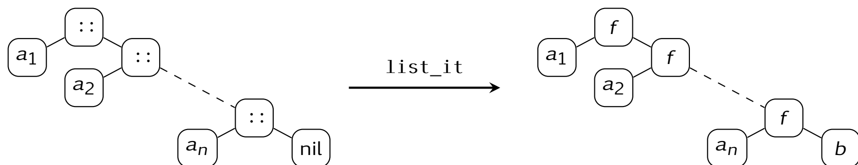
Exercice. Déterminer le rôle des fonctions suivantes :

- `let myst2 x lst = list_it (fun a b -> a::b) lst [x] ;;`

Fonctions d'ordre supérieur

La fonctionnelle list_it

Elle a pour objet d'effectuer la transformation :



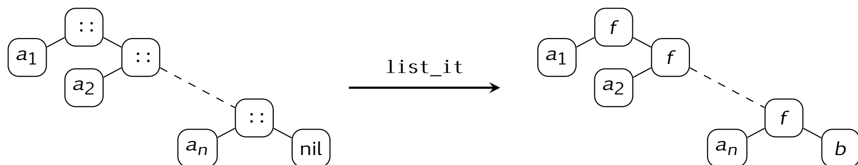
Exercice. Déterminer le rôle des fonctions suivantes :

- `let myst2 x lst = list_it (fun a b -> a::b) lst [x] ;;`
  → Insertion de l'élément **x** en queue de la liste **lst**.
- `let myst3 = list_it (fun a b -> a::b) ;;`

Fonctions d'ordre supérieur

La fonctionnelle list_it

Elle a pour objet d'effectuer la transformation :



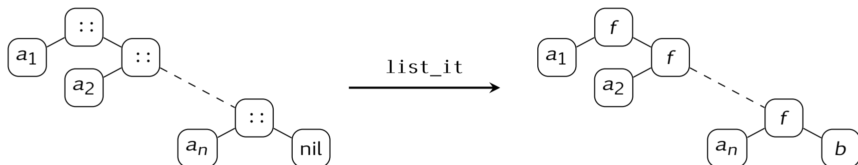
Exercice. Déterminer le rôle des fonctions suivantes :

- **`let myst2 x lst = list_it (fun a b -> a::b) lst [x] ;;`**
  $\longrightarrow$ Insertion de l'élément **`x`** en queue de la liste **`lst`**.
- **`let myst3 = list_it (fun a b -> a::b) ;;`**
  C'est équivalent à
  `let myst2 l1 l2 = list_it (fun a b -> a::b) l1 l2 ;;`
  $\longrightarrow$ Concaténation des deux listes **`l1`** et **`l2`**.

Fonctions d'ordre supérieur

La fonctionnelle list_it

Elle a pour objet d'effectuer la transformation :



La fonction **list_it** se définit ainsi :

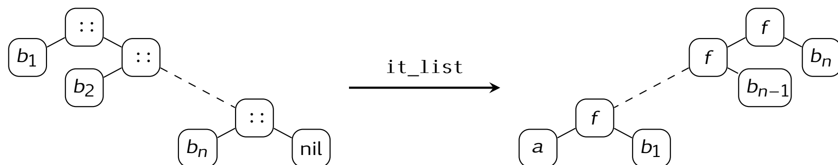
```
let rec list_it f lst b = match lst with        (* prédéfinie *)
  | []    -> b
  | t::q -> f t (list_it f q b) ;;
```

On la nomme en général *fold right*.

Fonctions d'ordre supérieur

La fonctionnelle it_list

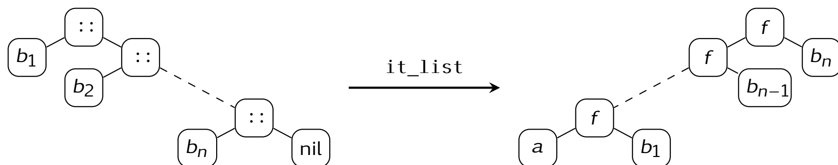
Elle a pour objet d'effectuer la transformation :



Fonctions d'ordre supérieur

La fonctionnelle it_list

Elle a pour objet d'effectuer la transformation :



On peut redéfinir la fonction **do_list** en posant :

```
let do_list f = it_list (fun a b -> f b) () ;;
```

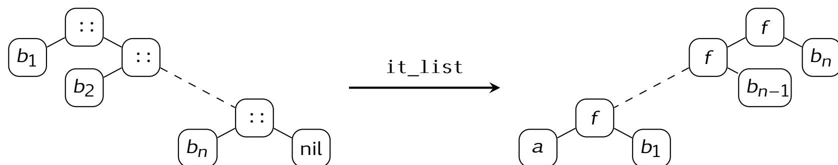
et redéfinir de nouveau la fonction **list_length** :

```
let list_length = it_list (fun a b -> a + 1) 0 ;;
```

Fonctions d'ordre supérieur

La fonctionnelle it_list

Elle a pour objet d'effectuer la transformation :



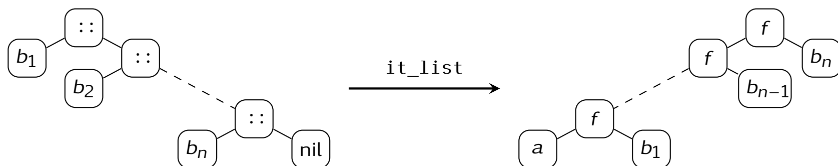
Exercice. Déterminer le rôle des fonctions suivantes :

- `let myst4 = it_list (prefix ^) "" ;;`

Fonctions d'ordre supérieur

La fonctionnelle it_list

Elle a pour objet d'effectuer la transformation :



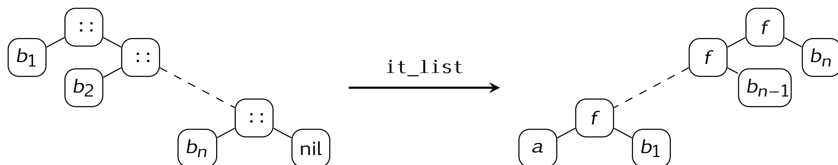
Exercice. Déterminer le rôle des fonctions suivantes :

- `let myst4 = it_list (prefix ^) "" ;;`
  → Concaténation des chaînes d'une *string list* .
- `let myst5 l = it_list min (hd l) (tl l) ;;`

Fonctions d'ordre supérieur

La fonctionnelle it_list

Elle a pour objet d'effectuer la transformation :



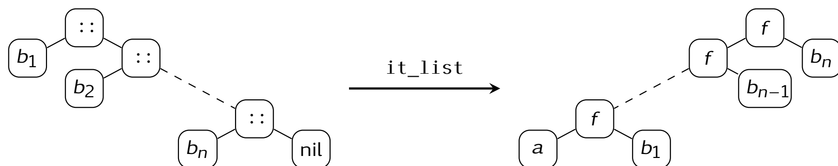
Exercice. Déterminer le rôle des fonctions suivantes :

- `let myst4 = it_list (prefix ^) "" ;;`
  → Concaténation des chaînes d'une *string list* .
- `let myst5 l = it_list min (hd l) (tl l) ;;`
  → Calcul de la valeur minimale d'une liste non vide.

Fonctions d'ordre supérieur

La fonctionnelle it_list

Elle a pour objet d'effectuer la transformation :



La fonction **it_list** se définit ainsi :

```
let rec it_list f a = function        (* prédéfinie *)
  | []   -> a
  | t::q -> it_list f (f a t) q ;;
```

On la nomme en général *fold left*.