Chapitre 2

Listes

1. Introduction

En informatique, une *structure de données* 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. Nous allons en étudier plusieurs, en commençant par les *listes simplement chainées* (ou plus simplement dans la suite de ce cours, les *listes*).
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. On peut donc représenter une liste L par la figure suivante :

$$\mathrm{L} \longrightarrow \boxed{a_0} \longrightarrow \boxed{a_1} \longrightarrow \boxed{a_2} \longrightarrow \cdots \longrightarrow \boxed{a_n} \longrightarrow \cdots$$

Néanmoins, un tel schéma est incomplet car taille arbitraire ne signifie pas taille infinie : il est nécessaire qu'une liste se termine. Il faut donc adjoindre à cette description un élément particulier caractérisant la terminaison de la liste, et qu'on appelle le *nil* (abréviation du latin *nihil*, autrement dit, rien).

$$\mathrm{L} \longrightarrow \boxed{a_0} \longrightarrow \boxed{a_1} \longrightarrow \boxed{a_2} \longrightarrow \cdots \longrightarrow \boxed{a_n} \longrightarrow \mathrm{nil}$$

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

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

• Mise en œuvre en Caml

Bien que cette structure de donnée soit déjà présente en Caml (et y joue un rôle primordial), nous allons définir un nouveau type qui implémente (si elle n'existait pas déjà) la structure de liste, pour en bien comprendre les avantages et les contraintes. Bien entendu cette définition n'est que provisoire ; une fois cette construction achevée nous nous empresserons de l'oublier pour ne plus utiliser que le type déjà défini en Caml.
On définit donc le type polymorphe et récursif *'a liste* :

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

Nous pouvons maintenant définir une première liste d'entiers, par exemple celle représentée par le schéma suivant :

$$\mathrm{lst} \longrightarrow \boxed{3} \longrightarrow \boxed{2} \longrightarrow \boxed{1} \longrightarrow \mathrm{nil}$$

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

Pour construire de nouvelles listes à partir de listes déjà créées, il peut être utile de posséder une fonction insérant un nouvel élément en tête de liste :

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

Pour définir la liste représentée ci-dessous, il suffit dès lors d'écrire :

$$\mathrm{lst2} \longrightarrow \boxed{4} \longrightarrow \boxed{3} \longrightarrow \boxed{2} \longrightarrow \boxed{1} \longrightarrow \mathrm{nil}$$

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

Inversement, il faut pouvoir récupérer les éléments d'une liste. Ceci nous amène à définir deux fonctions supplémentaires : une fonction **tete** qui retourne le premier élément d'une liste (si elle existe), et **queue** qui renvoie la liste pointée par la tête de liste :

```
# 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>
# tete lst2 ;;
- : int = 4
# queue lst2 ;;
- : int liste = Cellule (3, Cellule (2, Cellule (1, Nil)))
```

Nous n'allons pas pousser plus loin la mise en œuvre de ce type puisque le type *'a list* existe d'ores et déjà en Caml, mais on retiendra principalement de cette construction les observations suivantes :

- 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.

2. Description des listes en Caml

2.1 Construction d'une liste

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

```
# [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 est représenté par la liste vide `[]` et a pour type *'a list*.

L'ajout d'un élément en tête de liste est représenté par l'opérateur infixe `::` qu'on appelle *conse* (pour constructeur de liste) :

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

À l'inverse, les fonctions **hd** (*head*) et **tl** (*tail*) permettent d'obtenir la tête (le premier élément de la liste) et la queue (la liste privée de son premier élément) d'une liste :

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

Notons que ces deux fonctions déclenchent une exception lorsqu'on essaye de les appliquer à la liste vide :

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

2.2 Filtrage et récursivité

Aux caractères utilisables dans un motif évoqués au chapitre précédent viennent s'ajouter les caractères `[ ; ]` et `::`. Par exemple, le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prendra la valeur de la tête et **`q`** celle de la queue. Il est donc facile de redéfinir à titre d'exemple les fonctions **`hd`** et **`tl`** :

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

On prendra bien note que ces deux fonctions sont de coût constant.

Exercice. Décrire en français courant les listes reconnues par les motifs ci-dessous :

`[x]` `x::[]` `[]::x` `[1; 2; x]` `1::2::[x]` `1::2::x` `x::y::z`

Associé à la récursivité, le filtrage est le mode principal de définition d'une fonction agissant sur les listes. À titre d'exemple, nous allons passer en revue quelques exemples de fonctions agissant sur les liste, la plus-part étant prédéfinies dans le langage.

Calcul de la longueur d'une liste

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

Cette fonction a une complexité en $\Theta(\ell)$, où ℓ désigne la longueur de la liste : en effet, si $C(\ell)$ désigne cette complexité, on dispose de la relation : $C(\ell) = C(\ell-1)+\Theta(1)$ qui conduit par télescopage à $C(\ell) = \Theta(\ell)$.

Obtention du dernier élément d'une liste

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

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

Test d'appartenance à une liste

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

Cette fonction a une complexité en $O(\ell)$, où ℓ désigne la longueur de la liste.
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 ;;
```

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 ;;
```

Cette fonction a une complexité en $\Theta(\min(n,\ell))$, où ℓ désigne la longueur de la liste. On voit là une différence fondamentale avec le type *list* du langage Python pour lequel l'accès aux éléments est de coût constant.

Concaténation de deux listes

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

Cette fonction a une complexité en $\Theta(\ell_1)$ où ℓ_1 désigne la longueur de la liste de gauche.

Caml dispose de l'opérateur `@` qui est la forme infixe de cette fonction :

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

Remarque. Le mot-clé **prefix** permet de transformer un opérateur infixe en opérateur préfixe ; ainsi la fonction **concat** que nous venons de définir est équivalente à la fonction **prefix @**.

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

3. Fonctionnelles agissant sur les listes

Les fonctions que nous allons étudier maintenant sont un peu plus générales que les précédentes : leur usage au sein d'un code permet d'en simplifier l'écriture et par là même la compréhension.

3.1 Les fonctions map et do_list

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** a pour objet de créer la liste $[f(a_0); \cdots; f(a_{n-1})]$. Sa définition est la suivante :

```
let rec map f = function            (* fonction prédéfinie en Caml *)
  | []   -> []
  | t::q -> (f t)::(map f q) ;;
```

Son type est *('a –> 'b) –> 'a list –> 'b list* et sa complexité est en $\Theta(\ell)$, où ℓ désigne la longueur de la liste, si la complexité de la fonction f est constante.

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

Exercice. Déterminer le type et ce que réalise la fonction :

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

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** a pour objet d'effectuer la séquence $f(a_0); f(a_1); \ldots; f(a_{n-1})$ (ce qui n'a d'intérêt que si f a un effet sur l'environnement). Sa définition est la suivante :

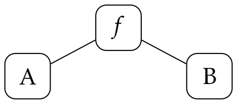
```
let rec do_list f = function        (* fonction prédéfinie en Caml *)
  | []   -> ()
  | t::q -> f t ; do_list f q ;;
```

Son type est *('a –> 'b) –> 'a list –> unit* et sa complexité est en $\Theta(\ell)$, où ℓ désigne la longueur de la liste, si la complexité de la fonction f est constante.

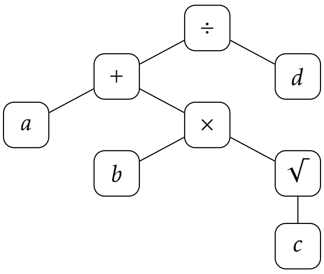
```
# do_list print_string ["alpha"; "beta"; "gamma"; "delta"] ;;
alphabetagammadelta- : unit = ()
```

3.2 Vers une tentative de généralisation : les fonctions list_it et it_list

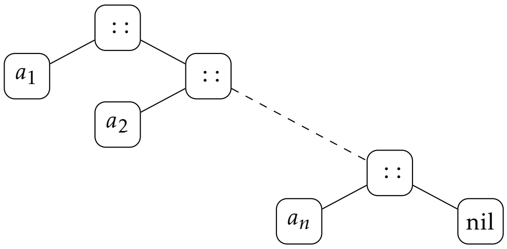
On conviendra aisément que les fonctions écrites jusqu'à présent suivent toutes peu ou prou le même schéma : un filtrage de la liste vide et une fonction à deux arguments avec pour premier la tête de la liste et pour second un appel récursif sur la queue. On peut donc envisager de généraliser cette situation. Pour ce faire, nous allons nous inspirer de la *représentation arborescente* d'une expression : sans rentrer dans les détails (nous y reviendrons au chapitre suivant), disons que les valeurs seront représentées par des feuilles et l'expression $f(a,b)$ par l'arbre :



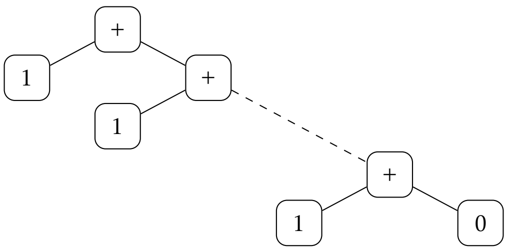
où A et B sont eux-même des arbres représentant respectivement les expressions a et b.
Par exemple, l'expression $\dfrac{a+b\sqrt{c}}{d}$ sera représentée par l'arbre :



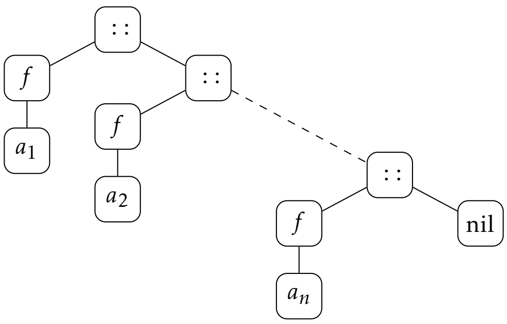
Ainsi, une liste $[a_1;a_2;\cdots;a_n]$ peut être représentée par l'arbre suivant :



On peut alors observer que calculer la longueur de cette liste revient à transformer l'arbre précédent en :

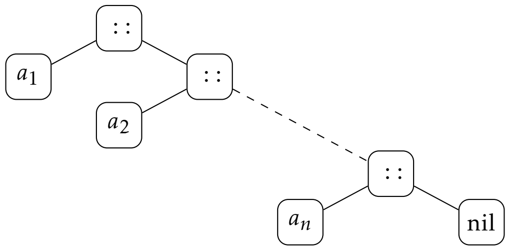


ou encore que lui appliquer la fonction `map f` revient à le transformer en :

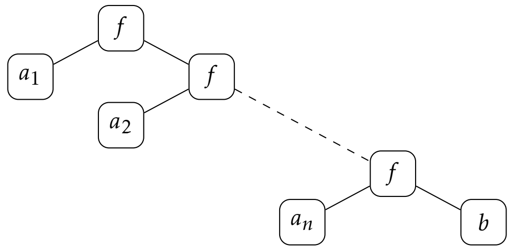


• La fonction list_it

Il existe une fonction Caml qui réalise de telles transformations : la fonction **list_it**, dont le type est : *('a –> 'b –> 'b) –> 'a list –> 'b –> 'b*. Elle a pour objet, étant donnés une fonction $f : \mathrm{A} \times \mathrm{B} \to \mathrm{B}$, une liste $[a_1; \cdots; a_n]$ d'éléments de A et un élément $b \in \mathrm{B}$, de calculer : $f\Big(a_1, f\Big(a_2, f\Big(a_3 \ldots, f(a_n, b)\Big)\Big)\Big)$, c'est à dire d'effectuer la transformation schématisée par :



list_it $\longrightarrow$



On peut alors 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 [] ;;
```

Exercice. Deviner ce que réalisent les fonctions définies ci-dessous :

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

b) `let myst3 = list_it (fun a b -> a::b) ;;`

La fonction **list_it** elle-même n'est pas compliquée à définir :

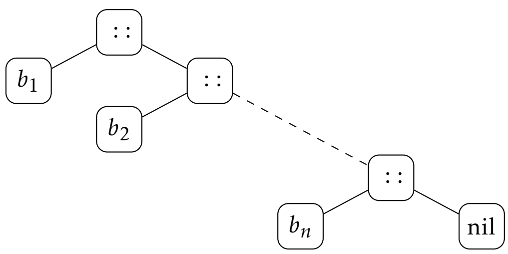
```
let rec list_it f lst b = match lst with        (* fonction prédéfinie en Caml *)
  | []   -> b
  | t::q -> f t (list_it f q b) ;;
```

• La fonction it_list

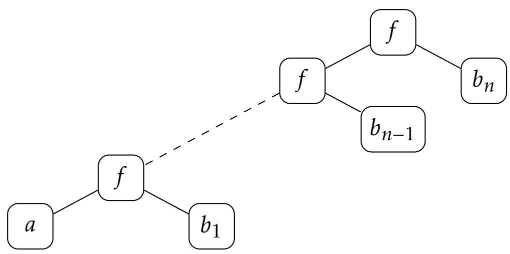
La fonctionnelle **it_list** a pour type *('a –> 'b –> 'a) –> 'a –> 'b list –> 'a* et a pour objet, étant donnés une fonction $f : \mathrm{A} \times \mathrm{B} \to \mathrm{A}$, un élément $a \in \mathrm{A}$ et une liste $[b_1; \cdots; b_n]$ d'éléments de B, de calculer

$$f\Big(\ldots f\Big(f\Big(f(a, b_1), b_2\Big), b_3\Big)\ldots, b_n\Big),$$

c'est à dire d'effectuer la transformation schématisée par :



it_list $\longrightarrow$



On peut par exemple redéfinir la fonction **do_list** de la manière suivante :

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

ou définir de nouveau la fonction **list_length** :

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

Exercice. Deviner ce que réalisent les fonctions définies ci-dessous :

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

b) `let myst5 l = it_list min (hd l) (tl l) ;;`

La fonction **`it_list`** se définit très simplement de la manière suivante :

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

Remarque. Ces deux fonctions **`list_it`** et **`it_list`** sont des fonctions génériques qui existent dans la plupart des langages fonctionnels (où elles sont en général connues sous le nom de *fold right* et *fold left*). Nous aurons l'occasion de constater que la majorité des fonctions agissant sur les listes suivent un schéma récursif semblable, et qu'en conséquence de quoi peuvent être définies à l'aide de l'une de ces fonctions (voire les deux). L'utilisation de ces fonctions génériques facilite ainsi la preuve de validité des programmes. Elles ne sont pas d'un abord facile mais leur bon usage permet de notablement simplifier certains codes.

4. Exercices

4.1 Parcours d'une liste

Exercice 1 Écrire une fonction qui retourne l'avant-dernier élément d'une liste, s'il existe.

Exercice 2 En procédant par récursivité et filtrage, définir une fonction calculant la somme des éléments d'une liste d'entiers.
En utilisant l'opérateur **`it_list`**, définir une fonction calculant le produit des éléments d'une liste d'entiers.

Exercice 3 Les fonctions suivantes sont prédéfinies en Caml. En procédant par récursivité et filtrage, en donner la définition.

a) **`exists`** est une fonction de type *('a –> bool) –> 'a list –> bool* qui détermine s'il existe un élément de la liste vérifiant une propriété donnée.

b) **`for_all`** est une fonction de type *('a –> bool) –> 'a list –> bool* qui détermine si tous les éléments de la liste vérifient une propriété donnée.

Les définir ensuite à l'aide de l'opérateur **`it_list`**.

Exercice 4 Rédiger une fonction Caml calculant la liste de tous les préfixes d'une liste donnée. Par exemple :

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

Exercice 5 Étant données une fonction de type *'a –> bool* et une liste de type *'a list*, définir une fonction calculant :

a) le n^e élément de la liste vérifiant la propriété (s'il en existe) ;

b) le dernier élément de la liste vérifiant cette propriété.

Exercice 6 L'image *miroir* d'une liste $[a_1;a_2\cdots;a_n]$ est la liste $[a_n;a_{n-1};\cdots;a_1]$ dans laquelle l'ordre des éléments a été inversé.

a) En procédant par récursivité et filtrage, définir une fonction **`rev`** réalisant cette transformation (vous pouvez utiliser l'opérateur de concaténation `@`). Montrer que le coût de cette fonction est quadratique.

b) Définir maintenant la fonction **`rev`** en utilisant l'opérateur **`it_list`**. En évaluer le coût.

c) Rédiger enfin une troisième version de cette fonction, de coût linéaire, et n'utilisant pas l'opérateur **`it_list`**.

Notez que cette fonction est prédéfinie en Caml.

Exercice 7 Écrire une fonction `rotg` qui fait tourner une liste d'un cran vers la gauche. Par exemple, `rotg [1; 2; 3; 4]` renverra la liste `[2; 3; 4; 1]`. Quel est le coût de votre algorithme ?

Mêmes questions pour la fonction `rotd` qui tourne une liste d'un cran vers la droite.

4.2 Insertion et suppression

Exercice 8 Définir une fonction de type *int –> 'a list –> 'a list* qui supprime le n^e élément d'une liste, puis une fonction de type *'a –> int –> 'a list –> 'a list* qui insère un élément dans une liste à la n^e position.

Exercice 9 Déterminer le type et préciser le rôle de la fonction suivante :

```
let rec myst f = function
  | []            -> []
  | t::q when f t -> t::(myst f q)
  | _::q          -> myst f q ;;
```

Exercice 10 On souhaite écrire une fonction `purge` qui, appliquée à une liste, retourne une liste dans laquelle les doublons ont été éliminés (on rappelle que la fonction prédéfinie `mem` détermine si un élément appartient ou pas à une liste).

a) Écrire une première version de `purge` dans laquelle seule la dernière occurrence de chaque doublon sera conservée. Par exemple, `purge [1; 2; 3; 1; 4; 3; 1]` renverra comme résultat `[2; 4; 3; 1]`.

b) Écrire une seconde version de `purge` dans laquelle seule la première occurrence de chaque doublon sera conservée. Cette fois, `purge [1; 2; 3; 1; 4; 3; 1]` renverra le résultat `[1; 2; 3; 4]`.

Exercice 11 On souhaite représenter un ensemble par une liste, chaque élément de l'ensemble ne devant apparaître qu'une seule fois dans la liste, à un emplacement arbitraire.

a) Définir une fonction `intersection` qui calcule l'intersection de deux ensembles. Évaluer son coût en fonction des cardinaux de ces ensembles.

b) Définir de même l'union et la différence symétrique de deux ensembles.

c) Rédiger enfin une fonction `egal` qui détermine si deux ensembles sont égaux.

Évaluer le coût de chacune de ces fonctions.