Listes

Table des matières

Les *structures de données* jouent un rôle essentiel en informatique. Elles permettent d'organiser une *collection* d'objets afin de les traiter efficacement. Nous avons déjà étudié en C la structure de *tableau* qui décrit un ensemble ordonné de n éléments du même type, chaque élément étant identifié par un index $k \in [\![0, n[\![$ et accessible avec une complexité temporelle en $\Theta(1)$. Les tableaux ont une taille n fixée lors de leur création et n'évoluant pas dans le temps. Cependant, il est souvent nécessaire de disposer de structures de données pour lesquelles on peut ajouter et retirer des éléments de manière efficace. Les *listes* vont répondre à ce problème. Ce sont des collections ordonnées d'éléments de même type pour lesquelles il est possible d'ajouter et de supprimer des éléments de manière efficace. On souhaite toujours pouvoir itérer rapidement sur leurs éléments, mais on va devoir renoncer à certaines propriétés des tableaux, notamment le fait d'accèder au k-ième élément en $\Theta(1)$.

Il existe de nombreuses manières d'implémenter des listes. Elles diffèrent par la disponibilité ou non de certaines opérations élémentaires et de leur complexité temporelle. Les deux solutions les plus courantes sont les *listes chainées* et les *tableaux redimensionnables*. Les premières sont très utilisées en programmation fonctionnelle, tandis que les seconds sont essentiels en programmation impérative.

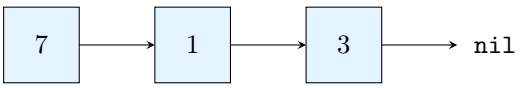
1 Liste chainée en OCaml

1.1 Le type `'a list`

Commençons par définir en OCaml la liste formée des entiers 7, 1 et 3. On écrit pour cela :

```
# let lst = [7; 1; 3];;
val lst : int list = [7; 1; 3]
```

OCaml a ainsi créé un objet de type `int list` que l'on se représente de la manière suivante.



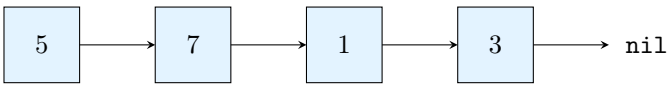
On dit que 7 est la *tête* de notre liste et que sa *queue* est la liste $[1; 3]$. Il est possible d'obtenir la tête et la queue d'une liste non vide à l'aide des fonctions `List.hd` (pour *head*) et `List.tl` (pour *tail*).

```
# List.hd lst;;
- : int = 7
# List.tl lst;;
- : int list = [1; 3];;
```

L'ajout d'un élément en tête de liste se fait avec l'opérateur infixe `::` qu'on appelle *conse* (pour *constructeur de liste*). Ainsi

```
# 5 :: lst;;
- : int list = [5; 7; 1; 3]
```

permet ainsi de construire la liste



dont la tête est l'entier 5 et la queue est la liste `lst`. Le conse et ce lien, entre la tête et la queue, qui est symbolisé par la flèche sur le schéma ci-dessus. La liste vide, notée `[]` est appelée *nil*. Une liste possédant un unique élément est obtenue en ajoutant ce dernier en tête de la liste vide.

```
# 9 :: [];;
- : int list = [9]
```

Il est d'ailleurs possible de construire n'importe quelle liste à l'aide de la liste vide et de l'opérateur conse.

```
# 7 :: (1 :: (3 :: []));;
- : int list = [7; 1; 3]
```

Les parenthèses sont facultatives et on aurait pu aussi bien écrire `7 :: 1 :: 3 :: []`. On comprend ainsi la raison de la présence d'une flèche vers le symbole `nil` dans nos schémas. On retiendra qu'une liste est soit vide, soit composée d'une tête et d'une queue.

Une liste sera toujours constituée d'éléments du même type. Une liste dont les éléments sont de type `'a` est de type `'a list`. Notons au passage que le type de la liste vide est `'a list` car le type de ses éléments n'est pas connu. On parle de *type paramétré*.

```
# ['o'; 'c'; 'a'; 'm'; 'l'];;
- : char list = ['o'; 'c'; 'a'; 'm'; 'l']
# [('a', 3); ('b', 1); ('e', 7)];;
- : (char * int) list = [('a', 3); ('b', 1); ('e', 7)]
# [[3; 7]; []; [9; 5; 2]];;
- : int list list = [[3; 7]; []; [9; 5; 2]]
# [];;
- : 'a list = []
```

1.2 Fonction et filtrage de motif

Nous avons vu plus haut qu'une liste est soit vide, soit composée d'une tête et d'une queue. Cette remarque est à la base du *filtrage de motif*, une composante essentielle du langage OCaml que nous allons mettre en oeuvre ici pour les listes.

La fonction suivante permet de déterminer si une liste commence par un entier pair.

```
let f (lst : int list) : bool =
  match lst with
  | [] -> false
  | x :: xs -> x mod 2 = 0
```

Elle se lit simplement : si la liste est vide, la réponse est `false` ; sinon la liste possède une tête que l'on nomme `x` et une queue que l'on nomme `xs` et la réponse est `true` si x est pair et `false` sinon. Notons que les deux expressions situées à droite de la flèche doivent être du même type.

```
# f [];;
- : bool = false
# f [5; 4; 0];;
- : bool = false;;
# f [2; 4; 0];;
- : bool = true
```

En combinant le filtrage de motif et la programmation récursive, il devient facile d'écrire une fonction calculant la longueur d'une liste.

```
let rec longueur (lst : 'a list) : int =
    match lst with
    | [] -> 0
    | x :: xs -> 1 + longueur xs
```

Cette fonction est disponible dans la bibliothèque standard avec `List.length`.

```
# List.length [7; 1; 3];;
- : int = 3
```

Si l'on souhaite écrire une fonction calculant la somme des éléments d'une liste, il suffit d'écrire :

```
let rec somme (lst : int list) : int =
    match lst with
    | [] -> 0
    | x :: xs -> x + somme xs
```

Exercices 1

- ⇨ Écrire une fonction `membre (x : 'a) (lst : 'a list) : bool` déterminant si x est présent dans la liste *lst*.
- ⇨ Écrire une fonction `concat (lst1 : 'a list) (lst2 : 'a list) : 'a list` concatenant les deux listes passées en argument. Par exemple, `concat [3; 1] [2; 7; 5]` doit renvoyer `[3; 1; 2; 7; 5]`. *Notons que cette fonction est disponible en OCaml sous la forme de l'opérateur infixe* `@`.
- ⇨ Écrire une fonction `insere (x : int) (lst : int list) : int list` qui insère x dans une liste triée par ordre croissant.
- ⇨ Écrire une fonction `affiche (lst : int list) : unit` affichant les éléments de la liste *lst*. On rappelle que la fonction `print_int (x : int) : unit` permet d'afficher l'entier x.

Si l'on souhaite définir nous-même la fonction `List.hd`, nous remarquons que cette fonction ne peut pas renvoyer de valeur lorsqu'on lui passe en argument la liste vide. En OCaml, une façon de résoudre ce type de problème est de *lever une exception*. La manière la plus simple de faire cela est d'écrire `failwith` suivi d'une chaine de caractères. On écrit ainsi

```
let tete (lst : 'a list) : 'a =
  match lst with
  | [] -> failwith "La liste est vide"
  | x :: xs -> x
```

Notons que puisque la queue `xs` de la liste n'est pas utilisée à droite de la flèche, il est possible de la filtrer avec `_`. On pourra donc écrire `x :: _ -> x` pour le second filtrage.

Il est possible d'utiliser des filtres plus complexes. Par exemple, la fonction suivante renvoie le plus petit élément d'une liste non vide.

```
let rec minimum (lst : int list) : int =
  match lst with
  | [] -> failwith "La Liste est vide"
  | [x] -> x
  | x :: xs -> min x (minimum xs)
```

De manière générale, lorsqu'on effectue un filtrage de motif, OCaml essaie de filtrer la valeur, filtre après filtre. C'est le premier filtre qui laisse passer notre valeur qui est exécuté. La fonction suivante permet de renvoyer le second élément de la liste lorsque celle-ci possède au moins deux éléments.

```
let second (lst : 'a list) : 'a =
  match lst with
  | x :: y :: ys -> y
  | _ -> failwith "La liste est trop petite"
```

Notons que le premier filtrage peut s'écrire aussi `_ :: y :: _ -> y`. Un fitrage de la forme « `_ ->` », toujours placé en dernière position, permet de récupérer tous les cas qui n'ont pas été filtrés avant.

La fonction suivante permet de déterminer si une liste est croissante.

```
let rec est_croissante (lst : int list) : int =
  match lst with
  | x :: y :: ys -> (x <= y) && (est_croissante (y :: ys))
  | _ -> true
```

Un filtrage peut contenir des valeurs litérales et toute expression faisant intervenir des variables « fraiches » deux à deux distinctes ainsi que des constructeurs : , pour les tuples et :: pour les listes. OCaml teste un à un tous les filtrages possibles et renvoie la valeur associé au premier filtrage qui est un succès. Par exemple, la fonction suivante renvoie l'élément d'indice k de la liste, l'élément d'indice 0 étant l'élément de tête, l'élément d'indice 1 le second, etc.

```
let rec kth (k : int) (lst : 'a list) : 'a =
  match (k, lst) with
  | 0, x :: _ -> x
  | _, _ :: xs -> kth (k - 1) xs
  | _ -> failwith "Liste trop petite"
```

Exercices 2

⇨ Décrire en français les listes reconnues par les motifs ci-dessous.

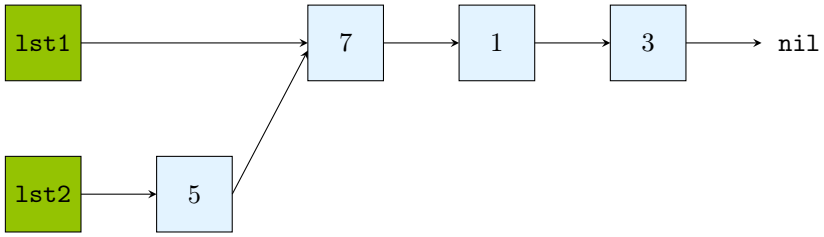
`[x]`, `x :: []`, `[] :: x`, `[1; 2; x]`, `1 :: 2 :: [x]`, `1 :: 2 :: x`, `x :: y :: z`

⇨ Écrire une fonction `doublet (lst : 'a list) : bool` déterminant si la liste *lst* possède deux fois la même valeur à la suite.

1.3 Modèle mémoire, immutabilité et complexité

Lorsqu'on crée une liste en ajoutant un élément de tête à une liste déjà existente, il est important de comprendre qu'on obtient une nouvelle liste qui partage ses éléments avec la liste d'origine.

```
# let lst1 = [7; 1; 3];;
val lst1 : int list = [7; 1; 3]
# let lst2 = 5 :: lst1;;
val lst2 : int list = [5; 7; 1; 3]
```

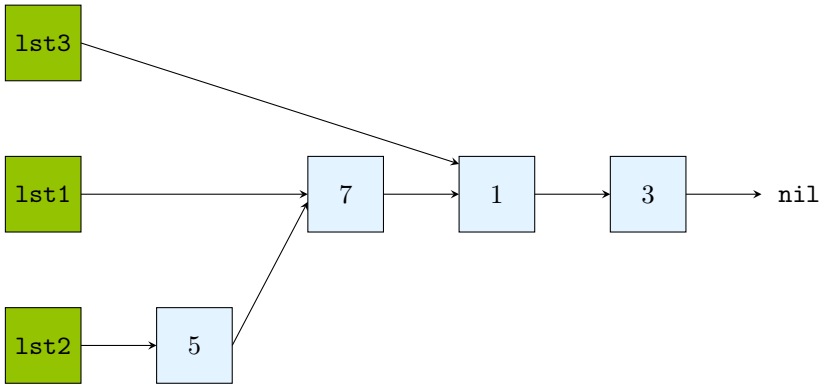


La création de la liste `lst2` a simplement nécessité le création d'une nouvelle cellule contenant 5 et d'une flèche vers la liste `lst1`. La complexité temporelle de cette opération est donc en $\Theta(1)$.

Si après ces opérations, on récupère la queue de la liste `lst1`

```
# let lst3 = List.tl lst1;;
val lst1 : int list = [1; 3]
```

on obtient le schéma suivant en mémoire.



On comprend donc pourquoi la fonction `List.tl` a une complexité temporelle en $\Theta(1)$. On retiendra donc que le constructeur `::`, les fonctions `List.hd`, `List.tl` et donc le filtrage de motif sont toutes des opérations de complexité temporelle $\Theta(1)$.

Ce partage en mémoire pourrait poser problème s'il était possible de modifier la liste `lst1`, par exemple en changeant le 7 en 9. En effet, du fait du partage en mémoire, la liste `lst2` serait alors changée en $[5;9;1;3]$. Cependant, il est impossible de modifier une liste : les listes sont des objets *immuables* en OCaml. C'est grâce à cette immutabilité que l'on peut tout à fait imaginer que les 3 listes construites dans cet exemple sont totalement indépendantes.

Ces remarques nous permettent de calculer la complexité temporelle des fonctions opérant sur les listes. Par exemple, si l'on considère la fonction

```
let rec longueur (lst : 'a list) : int =
  match lst with
  | [] -> 0
  | x :: xs -> 1 + longueur xs
```

permettant de calculer la longueur n d'une liste lst, on a :

$$c_0 = \Theta(1) \qquad \text{et} \qquad \forall n \in \mathbb{N}^*, \quad c_n = c_{n-1} + \Theta(1).$$

On en déduit que

$$\begin{aligned} c_n &= c_0 + \sum_{k=0}^{n-1} (c_{k+1} - c_k) \\ &= \Theta(1) + \sum_{k=0}^{n-1} \Theta(1) \\ &= \Theta(n) \end{aligned}$$

Le calcul de `longueur lst` se fait donc en complexité linéaire en la longueur de la liste `lst`. La fonction `List.length` possède d'ailleurs la même complexité.

Exercices 3

⇨ Calculer la complexité de la fonction

```
let rec somme (lst : int list) : int =
  match lst with
  | [] -> 0
  | x :: xs -> x + somme xs
```

calculant la somme des éléments d'une liste.

⇨ Calculer la complexité de la fonction

```
let rec concat (lst1 : 'a list) (lst2 : 'a list) : 'a list =
  match lst1 with
  | [] -> lst2
  | x :: xs -> x :: (concat xs lst2)
```

réalisant la concaténation de deux listes.

La concaténation de deux listes en OCaml est disponible avec l'opérateur infixe `@`.

```
# [5; 7] @ [1; 3];;
- : int list = [5; 7; 1; 3]
```

La complexité temporelle de `u @ v` est en $\Theta(|u|)$ où $|u|$ représente la longueur de la liste u.