Chapitre 2

Correction des exercices

Parcours d'une liste

Exercice 1 Le motif `[a]` est reconnu par toute liste de longueur 1, et `[a; b]` par toute liste de longueur 2.

```
let rec avant_dernier = function
  | [] | [_]  -> raise Not_found
  | [a; _]    -> a
  | _::q      -> avant_dernier q ;;
```

Exercice 2 En procédant par récursivité et filtrage on obtient :

```
let rec somme = function
  | []   -> 0
  | t::q -> t + somme q ;;
```

et avec l'opérateur **`it_list`** :

```
let somme = it_list (prefix +) 0 ;;
```

Pour le produit, c'est exactement la même chose : **`let produit = it_list (prefix *) 1 ;;`**

Exercice 3 On peut commencer par utiliser un motif gardé :

```
let rec exists prop = function
  | []                -> false
  | t::q when prop t -> true
  | _::q              -> exists prop q ;;
```

mais le principe de l'évaluation paresseuse nous permet aussi la rédaction suivante, sans perte de performance (comprenez-vous pourquoi ?) :

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

On peut aussi utiliser l'opérateur **`it_list`**, mais ici on perd le bénéfice de l'évaluation paresseuse :

```
let exists prop = it_list (fun a b -> a || prop b) false ;;
```

Pour le deuxième opérateur demandé, c'est exactement la même chose :

```
let rec for_all prop = function             (* fonction prédéfinie en Caml *)
  | []   -> true
  | t::q -> (prop t) && (for_all prop q) ;;
```

ou bien :

```
let for_all prop = it_list (fun a b -> a && prop b) true ;;
```

Exercice 4 Une solution consiste à utiliser **map** avec une fonction d'insertion en tête de liste :

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

Exercice 5 Calculer le n^e élément vérifiant une propriété donnée ne pose guère de problème en procédant par filtrage avec motif gardé :

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

En revanche, il est plus ardu de calculer le dernier élément de la liste vérifiant cette propriété. La solution consiste à utiliser une fonction auxiliaire à deux variables, l'une de ces deux variables servant d'*accumulateur* (de manière imagée, disons que cet accumulateur va se « souvenir » du dernier élément rencontré vérifiant la propriété) :

```
let last prop lst =
   let rec aux acc = function
     | []                    -> acc
     | t::q when not prop t -> aux acc q
     | t::q                  -> aux [t] q
   in
   match aux [] lst with
     | [] -> raise Not_found
     | x  -> hd x ;;
```

Une solution plus simple consiste à chercher le premier élément vérifiant la propriété de l'image miroir de la liste (calculée par la fonction **rev**, voir l'exercice suivant à son sujet) :

```
let last prop lst =
   let rec first = function
     | []               -> raise Not_found
     | t::q when prop t -> t
     | _::q             -> first q
   in first (rev lst) ;;
```

Exercice 6 La première solution est facile à écrire :

```
let rec miroir = function
  | []   -> []
  | t::q -> (miroir q) @ [t] ;;
```

mais n'est pas efficace. En effet, si on prend comme indice de performance le nombre d'insertion en tête de liste effectuées, le coût de l'opérateur de concaténation est égal à la longueur de la première liste. Le nombre u_n d'insertions effectué par la fonction précédente vérifie donc la relation de récurrence : $u_n = u_{n-1} + n - 1$, ce qui conduit à $u_n = \dfrac{n(n-1)}{2} = \Theta(n^2)$; le coût est quadratique.
En revanche, la définition suivante :

```
let miroir = it_list (fun a b -> b::a) [] ;;
```

a un coût linéaire. En effet, il est clair que la fonction **list_it** applique exactement n fois la fonction f (voir le schéma illustrant cette fonctionnelle), donc le nombre u_n d'insertions en tête de liste effectuée par la nouvelle fonction **miroir** va être égal à : $u_n = n$.
Pour obtenir une version de **miroir** n'utilisant pas la fonction **list_it**, commençons par définir une fonction auxiliaire ayant le même rôle que **it_list (fun a b -> b::a)**, en s'inspirant de la définition de **it_list** :

```
let miroir =
   let aux acc = function
     | []   -> acc
     | t::q -> aux (t::acc) q
   in aux [] ;;
```

Ce faisant, on peut constater que la première des deux variables de la fonction auxiliaire ne joue finalement qu'un rôle d'accumulateur, tout comme l'exercice précédent. Nous aurons l'occasion d'en reparler.

Exercice 7 La rotation à gauche est très simple et a un coût linéaire :

```
let rotg = function
  | []   -> []
  | t::q -> q @ [t] ;;
```

En revanche, il est plus délicat d'obtenir une rotation à droite de coût linéaire, sauf si on se souvient qu'on vient de constater dans l'exercice précédent que la fonction **rev** est de coût linéaire. Il suffit alors d'écrire :

```
let rotd lst = rev (rotg (rev lst)) ;;
```

Insertion et suppression

Exercice 8 La définition de ces deux fonctions ne pose guère de problème :

```
let rec supprime = fun
  | _ []     -> []
  | 1 (_::q) -> q
  | n (t::q) -> t::(supprime (n-1) q) ;;
```

(on convient que si n est supérieur à la longueur de la liste on ne fait rien).

```
let rec insere a = fun
  | 1 lst    -> a::lst
  | n []     -> failwith "liste trop courte"
  | n (t::q) -> t::(insere a (n - 1) q) ;;
```

Exercice 9 La fonction **myst** est de type *('a -> bool) -> 'a list -> 'a list*, et **myst prop lst** a pour objet de calculer la liste des éléments de la liste **lst** vérifiant la propriété **prop**.

Exercice 10 La première version s'écrit :

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

Sachant que le coût de la fonction **mem** est linéaire, le coût de cette fonction est quadratique dans le pire des cas (lorsque la liste ne contient aucun doublon).
La seconde version utilise une fonction auxiliaire pour éliminer les occurrences d'un élément dans une liste :

```
let rec elimine a = function
  | []              -> []
  | t::q when t = a -> elimine a q
  | t::q            -> t::(elimine a q) ;;

let rec purge = function
  | []   -> []
  | t::q -> t::(elimine t (purge q)) ;;
```

Sachant que le coût de la fonction **elimine** est à l'évidence linéaire, le coût de cette deuxième version de la fonction **purge** est là encore quadratique.

Exercice 11 Nous allons utiliser la fonction **mem** pour écrire les différentes fonctions qui nous sont demandées.

```
let rec intersection lst = function
  | []                  -> []
  | t::q when mem t lst -> t::(intersection lst q)
  | _::q                -> intersection lst q ;;
```

Sachant que le coût de la fonction **mem** est linéaire, le coût de cette fonction est quadratique.

```
let rec union lst = function
  | []                  -> lst
  | t::q when mem t lst -> union lst q
  | t::q                -> t::(union lst q) ;;
```

Là encore, le coût est quadratique.
Pour définir la différence symétrique, nous pouvons commencer par définir une fonction qui élimine l'(unique) occurrence d'un élément dans une liste quand ce dernier est présent :

```
let rec elimine a = function
  | []              -> []
  | t::q when t = a -> q                (* a est présent au plus une fois *)
  | t::q            -> t::(elimine a q) ;;

let rec difference lst = function
  | []                  -> lst
  | t::q when mem t lst -> difference (elimine t lst) q
  | t::q                -> t::(difference lst q) ;;
```

Le coût est une fois de plus quadratique.
Enfin, on peut remarquer que deux ensembles sont égaux si et seulement si leur différence symétrique est vide, ce qui permet de définir simplement :

```
let egal x y = (difference x y = []) ;;
```

à moins que l'on préfère :

```
let rec egal lst = function
  | []                  -> lst = []
  | t::q when mem t lst -> egal (elimine t lst) q
  | _                   -> false ;;
```

Chacune des deux solutions a un coût quadratique.