Programmation Fonctionnelle : TD3

Université de Tours

`Département informatique de Blois`

Récursivité et listes

*
* *

Appropriation du cours

Reprendre le cours 3 et tester en OCaml les exemples, en particulier vous devez très bien comprendre :

- La fonction `nbOccurences`
- La fonction `retireOccurence1`
- La fonction `plusGrand` (dans les deux versions, vérifier avec la trace que la version 2 entraîne moins de calculs).

Problème 1

1. Écrire la spécification puis le code d'une fonction `sommeListe1 l` qui retourne la somme des éléments d'une liste l d'entiers donnée en entrée.

$$\texttt{sommeListe1} : \begin{cases} \texttt{List} < \texttt{int} > & \rightarrow \texttt{int} \\ l = [x_1, ..., x_n] & \mapsto \sum_{i=1}^{n} x_i \end{cases}$$

```
let rec sommeList1 l = match l with
      []   -> 0
    | h::t -> h+sommeList1 t;;
```

2. Ecrire la spécification et le code d'une fonction `mulScalaire l lambda` qui multiplie par $\lambda \in \mathbb{R}$ chaque élément d'une liste l de réels donnée en entrée.

$$\texttt{mulScalaire} : \begin{cases} \texttt{List} < \texttt{double} > \times \texttt{double} & \rightarrow \texttt{List} < \texttt{double} > \\ l = [x_1, ..., x_n], \lambda & \mapsto l' = [x_1\lambda, ..., x_n\lambda], \end{cases}$$

```
let rec mulScalaire l lambda = match (l,lambda) with
      ([],lambda)   -> []
    | (h::t,lambda) -> (h*.lambda)::mulScalaire t lambda;;
```

Problème 2

1. Écrire la spécification puis le code d'une fonction `sommeListe2 l1 l2` qui prend en paramètres deux listes l'entiers l_1 et l_2 et retourne la liste des sommes des éléments de même rang sur la plus courte des deux longueurs de liste (càd : $\min(|l_1|, |l_2|)$).

$$\texttt{sommeListe2} : \begin{cases} \texttt{List} < \texttt{int} > \times \texttt{List} < \texttt{int} > & \to \texttt{List} < \texttt{int} > \\ l_1 = [x_1, ..., x_n], l_2 = [y_1, ..., y_p] & \mapsto l_3 = [z_0, ..., z_{\min(n,p)}] \end{cases}$$

avec $\forall i \in [\![0, \min(n,p)]\!], z_i = x_i + y_i$

```
let rec sommeListe2 l1 l2 = match (l1,l2) with
  ([],l2) -> []
| (l1,[]) -> []
| (h1::t1, h2::t2) -> (h1+h2)::sommeListe2 t1 t2;;
```

2. Donner une version `sommeListe3` où le résultat contient autant d'éléments que la plus longue des deux listes. Les éléments sans correspondance dans l'autre liste sont simplement recopiés à la fin.

$$\texttt{sommeListe3} : \begin{cases} \texttt{List} < \texttt{int} > \times \texttt{List} < \texttt{int} > & \to \texttt{List} < \texttt{int} > \\ l_1 = [x_1, ..., x_n], l_2 = [y_1, ..., y_p] & \mapsto l_3 = [z_0, ..., z_{\max(n,p)}] \end{cases}$$

avec $\forall i \in [\![0, \max(n,p)]\!], z_i = \begin{cases} x_i + y_i & \text{pour } 0 \leq i \leq \min(n,p) \\ \begin{cases} x_i & \text{si } n > p \\ y_i & \text{sinon} \end{cases} & \text{pour } i > \min(n,p) \end{cases}$

```
let rec sommeListe3 l1 l2 = match (l1,l2) with
  ([],[])   -> []
| ([],l2) -> l2
| (l1,[]) -> l1
| (h1::t1, h2::t2) -> (h1+h2)::sommeListe3 t1 t2;;
```

Problème 3

1. Écrire la spécification et le code d'une fonction `equal l1 l2` qui prend en entrée deux listes l_1 et l_2 et qui retourne vrai si et seulement l_1 et l_2 contiennent le même nombre d'éléments et que pour chaque indice, l'élément i de l_1 est égal à l'élément i de l_2.

$$\texttt{equal} : \begin{cases} \texttt{List} < \texttt{T} > \times \texttt{List} < \texttt{T} > & \to \{true, false\} \\ l_1 = [x_1, ..., x_n], l_2 = [y_1, ..., y_p] & \mapsto \begin{cases} true & \text{si } n = p \text{ et } \forall i \in [\![1, n]\!], x_i = y_i \\ false & \text{sinon} \end{cases} \end{cases}$$

```
let rec equal l1 l2 = match (l1,l2) with
  ([],[])   -> true
| ([],h::t) -> false
| (h::t,[]) -> false
| (h1::t1, h2::t2) -> h1 = h2 && (equal t1 t2);;
```

2. On donne la spécification suivante $\texttt{miroir} : \begin{cases} \texttt{List} < \texttt{T} > & \to \texttt{List} < \texttt{T} > \\ l = [x_0, ..., x_{n-2}, x_{n-1}] & \mapsto l' = [x_{n-1}, x_{n-2}, ..., x_0] \end{cases}$

   Écrire le code correspondant à `miroir l`. On pourra utiliser l'opérateur de concaténation `@` d'Ocaml.

```
let rec miroir l = match l with
  []     -> []
| (h::t) -> miroir(t)@[h];;
```

3. Un *palindrome* est un mot qui peut être lu dans les deux sens (de droit à gauche ou de gauche à droite). Par exemple : “rotor”, “ete”, “callac” ou bien “mon nom”.

   Écrire la spécification et le code d'une fonction `palindrome l` qui prend en entrée une liste de caractères l et retourne vrai si le mot formé par la liste est un palindrome et faux sinon.

```
let palindrome l = equal l (miroir l);;
```