TP n° 2 : Les listes

1 Les listes en CAML

1.1 Construction de listes

OCAML permet de manipuler des listes *homogènes*, c'est-à-dire constituées d'éléments du même type.

Les listes sont des structures de données récursives définies à partir de deux constructeurs : `[]` pour la liste vide, et `::` pour l'adjonction d'un élément en tête de liste.

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

Il est aussi possible de construire une liste par énumération de ses éléments sous la forme

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

Remarque : La construction d'une liste par énumération de ses éléments utilise le séparateur `;` à ne pas confondre avec le séparateur `,`.

```
# [1,2,3] ;;
- : (int * int * int) list = [(1, 2, 3)]
```

1.2 Le type `list`

À la différence des n-uplets, les listes de OCAML sont *homogènes*.

```
# [1; "a"; true] ;;
Characters 4-7:
  [1; "a"; true] ;;
      ^^^
This expression has type string but is here used with type int
```

OCAML précise le type d'une liste en indiquant σ `list` où σ est le type des éléments de la liste.

```
# [1 ; 2; 3] ;;
- : int list = [1; 2; 3]
# [true; false] ;;
- : bool list = [true; false]
# [(1,2) ; (3,4)];;
- : (int * int) list = [(1, 2); (3, 4)]
```

La liste vide peut être d'un type polymorphe si elle est construite avec le constructeur `[]`

```
# [] ;;
- : 'a list = []
```

ou d'un type instancié si elle est obtenue par à partir d'une autre liste

```
# List.tl [true] ;;
- : bool list = []
# List.tl [1] ;;
- : int list = []
# List.tl ["a"] ;;
- : string list = []
```

1.3 Fonctions du module `List`

Les principales opérations sur les listes sont regroupées dans le module `List`. Pour les utiliser, il faut préfixer leur nom par le nom du module `List`[1].

Voici une liste non exhaustive des fonctions du module `List`.

- `hd : 'a list -> 'a` donne le premier élément de la liste (la tête de la liste). **CU :** la liste ne doit pas être vide.

```
# List.hd [1;2;3;4] ;;
- : int = 1
# List.hd [] ;;
Exception: Failure "hd".
```

- `tl : 'a list -> 'a list` donne la liste sans son premier élément (le reste de la liste). **CU :** la liste ne doit pas être vide.

```
# List.tl [1;2;3;4] ;;
- : int list = [2; 3; 4]
# List.tl [] ;;
Exception: Failure "tl".
```

- `length : 'a list -> int` donne la longueur de la liste.

```
# List.length [1;2;3;4] ;;
- : int = 4
# List.length [] ;;
- : int = 0
```

- `nth : 'a list -> int -> 'a` donne l'élément d'indice n de la liste, le premier élément étant d'indice 0. **CU :** n doit être inférieur à la longueur de la liste.

```
# List.nth [1;2;3;4] 0 ;;
- : int = 1
# List.nth [1;2;3;4] 3 ;;
- : int = 4
# List.nth [1;2;3;4] 4 ;;
Exception: Failure "nth".
```

- `append : 'a list -> 'a list -> 'a list` donne la liste concaténée des deux listes. L'opérateur `@` peut aussi être utilisé à la place de cette fonction.

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

- `map : ('a -> 'b) -> 'a list -> 'b list` applique une fonction à tous les éléments d'une liste pour donner la liste des résultats.

```
# List.map (function x -> x*x) [1;2;3;4] ;;
- : int list = [1; 4; 9; 16]
```

- `flatten : 'a list list -> 'a list` applatit la liste passée en paramètre.

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

2 Filtrage de motifs sur les listes

On l'a déjà signalé, les listes sont des structures de données récursives définies à partir de deux constructeurs : `[]` pour la liste vide, et `::` pour l'adjonction d'un élément en tête de liste.

Pour sélectionner un élément particulier de la liste, on peut bien entendu utiliser la fonction `nth` du module `List`. Mais on peut aussi utiliser du filtrage.

Voici par exemple programmées par filtrage de motifs les fonctions `tete` et `reste` équivalentes aux fonctions `hd` et `tl` du module `List`. Ces deux fonctions déclenchent une exception lorsque la liste est vide.

1. à moins de faire appel préalablement à la commande **open** `List ;;`

```
let tete = function
  | x::_ -> x
  | _ -> failwith "tete : liste vide"

let reste = function
  | _::l -> l
  | _ -> failwith "tete : liste vide"
```

Le calcul de la longueur d'une liste avec un filtrage de motifs se programme simplement.

```
let rec longueur = function
  | _::l -> 1 + longueur l
  | _ -> 0
```

On peut réécrire ces fonctions sans utiliser le mot **function** mais en utilisant l'expression `match ... with`. Voici par exemple la fonction longueur.

```
let rec longueur l =
  match l with
  | _::l -> 1 + longueur l
  | _ -> 0
```

C'est avec l'expression `match ... with` que l'on peut filtrer les arguments d'une fonction sous forme curryfiée. Voici l'exemple d'une fonction qui construit une liste d'entiers obtenus en ajoutant les entiers de même rang des deux listes passées en paramètre.

```
let rec liste_sommes l1 l2 =
  match (l1,l2) with
   | ([],[]) -> []
   | (x1::l1',x2::l2') -> (x1+x2)::(liste_sommes l1' l2')
   | _ -> failwith "liste_sommes : liste de longueurs differentes"
```

3 Petits exercices

Exercice 1 *Fonctions du module* `List`

Programmez les fonctions `nth`, `append`, `flatten` et `map` du module `List` en n'utilisant que du filtrage de motifs.

Exercice 2

Si `l` est une `'a list`, que représente la déclaration

```
let x = nth l
```

Exercice 3

Question 1 Programmez la fonction `somme` de type `int list` $\rightarrow$ `int` qui calcule la somme des éléments de la liste.

Question 2 Utilisez `map` et `somme` pour programmer la fonction `length` de type `'a list` $\rightarrow$ `int`.

Exercice 4 *Liste d'itérés*

Question 1

Programmez une fonction qui prend pour paramètres

1. une fonction f de type `'a` $\rightarrow$ `'a`,
2. un élément x de type `'a`,
3. et un entier $n \geq 0$,

qui construit la liste $[x;\ f(x);\ f^2(x);\ \ldots\ ;f^n(x)]$.

Question 2 Utilisez cette fonction pour définir une fonction à un argument entier $n \geq 0$ qui produit la liste des entiers consécutifs de 0 à n.