Définitions de type

Les types énumérés

La définition d'un type se fait à l'aide du mot clé `type`. Les noms des constructeurs commencent par une majuscule. Un constructeur permet de construire les valeurs d'un type et d'accéder aux composantes de ces valeurs grâce au filtrage des motifs.

```
# type couleur = Bleu | Rouge | Vert;;
type couleur = Bleu | Rouge | Vert
# Vert;;
- : couleur = Vert
# let valeur = function
   Bleu  -> 1
 | Vert  -> 2
 | Rouge -> 3;;
val valeur : couleur -> int = <fun>
# valeur Vert;;
- : int = 2
```

La somme de deux types

La somme de deux types correspond à l'union disjointe.

```
# type int_ou_bool = I of int | B of bool;;
type int_ou_bool = I of int | B of bool

# I;;

The constructor I expects 1 argument(s),
but is here applied to 0 argument(s)
```

```
# I 4;;
- : int_ou_bool = I 4
# B true;;
- : int_ou_bool = B true

# B 5;;

This expression has type int
but is here used with type bool
```

L'utilisation des expression de ce type se fait par "pattern matching".

```
# let f = function
    B x -> if x then 1 else 2
|   I x -> x+4;;
val f : int_ou_bool -> int = <fun>
# let f' = function
    B true -> 1
|   B false -> 2
|   I x -> x+4;;
val f' : int_ou_bool -> int = <fun>

# type string_ou_intlist =
         S of string | IL of int list;;
type string_ou_intlist = S of string | IL of int list

# type ('a,'b) somme = G of 'a | D of 'b;;
type ('a, 'b) somme = G of 'a | D of 'b

# G 1;;
- : (int, 'a) somme = G 1
# G true;;
- : (bool, 'a) somme = G true
# D [1; 2];;
- : ('a, int list) somme = D [1; 2]
```

Les listes

Des valeurs d'un même type peuvent être regroupées en des listes. Le type `list` est en fait un schéma de type prédéfini. La plupart des fonctions de manipulation des listes sont définies dans la bibliothèque `List`.

La liste vide est une valeur non fonctionnelle qui est de type polymorphe. Objective Caml permet de générer des listes d'objets de même type sans que ce type soit précisé. Ce polymorphisme est appelé *paramétrique*.

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

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

On note `[e1;...;en]` la liste `e1::(e2::...  (en::[])...)`.

```
# [[1;2];[];[6];[7;8;5;2]];;
- : int list list = [[1; 2]; []; [6]; [7; 8; 5; 2]]
```

La "déstructuration" des listes (extraction des éléments) se fait par "pattern-matching".

```
# let f = function
            [] -> 0
        |  x::l -> x+1;;
val f : int list -> int = <fun>
# f [];;
```

```
- : int = 0
# f [3;2;7];;
- : int = 4

# let f = function
            [] -> 0
          |  []::l -> 1
          |  (x::l)::m -> x+1;;
val f : int list list -> int = <fun>
# f [ []; [1;2] ];;
- : int = 1
# f [ [3;2]; []; [1;4] ];;
- : int = 4
```

Fonctions récursives sur les listes

Dans l'exemple suivant on calcule la somme des éléments d'une liste d'entiers.

```
# let rec somme = function
        [] -> 0
  |  x::l -> x+somme l;;
val somme : int list -> int = <fun>
# somme [3;2;7];;
- : int = 12
```

Dans l'exemple suivant on calcule la conjonction des éléments d'une liste de booléens.

```
# let rec conj = function
        [] -> true
  |  x::l -> x & conj l;;
val conj : bool list -> bool = <fun>
# conj [true; false; true];;
- : bool = false
```

```
# let rec longueur = function
      [] -> 0
  | x::l -> 1 + longueur l;;
val longueur : 'a list -> int = <fun>
# longueur [true; false];;
- : int = 2
# longueur [[1];
```

On peut également utiliser la fonction `tl` (tail) de la librairie `List`

```
# let rec longueur l =
    if l=[] then 0
    else 1 + longueur (List.tl l)
;;
val longueur : 'a list -> int = <fun>
```

On peut enfin utiliser la fonction `length` de la bibliothèque `List`. Attention elle prend un temps proportionnel à la longueur de la liste.

```
# List.length [1;3;5];;
- : int = 3
# List.hd [1;3;5];;
- : int = 1
```

La fonctionnelle `map`

Considérons l'exemple suivant.

```
# let rec succl=function
      [] -> []
  | x::l -> (x+1)::succl l;;
val succl : int list -> int list = <fun>
# succl [2;1;5];;
- : int list = [3; 2; 6]
```

Voici maintenant une fonction qui à une liste d'entiers $[n_1; \ldots, n_k]$ associe la liste de booléens $[b_1; \ldots; b_k]$ telle que b_i soit `true` si et seulement si n_i est pair.

```
let rec pairl = function
      [] -> []
  | x::l -> ((x mod 2)=0)::pairl l;;
val pairl : int list -> bool list = <fun>
# pairl [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```

La fonctionnelle `map` est définie dans la librairie `List`. Son type est

```
# List.map;;
- : f:('a -> 'b) -> 'a list -> 'b list = <fun>
```

et son effet est le suivant.

$$\texttt{List.map f}\,[e_1; \ldots; e_n] = [\texttt{f}\,e_1; \ldots; \texttt{f}\,e_n]$$

Cette fonction peut aussi s'écrire de la façon suivante.

```
# let rec map f = function
      [] -> []
  | x::l -> (f x)::map f l
```

```
;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

On peut alors réécrire les expressions vues plus haut en

```
# List.map (function x->x+1) [3; 2; 6];;
- : int list = [4; 3; 7]
# List.map (function x->(x mod 2)=0) [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```

Exercice Synthétiser le type de `map`.

Fonctionnelles d'itérations sur les listes

Il y a en Objective Caml deux fonctionnelles d'itération `fold_left` et `fold_right` définies dans la librairie `List`. Ces deux fonctionnelles permettent l'écriture compacte d'une fonction de calcul sur les éléments d'une liste comme la somme des éléments.

Leurs effets sont les suivants.

$$\texttt{fold\_right f [}e_1\texttt{;}e_2\texttt{;}\ldots\texttt{;}e_n\texttt{]}\ a \;=\; (\texttt{f}\ e_1\ (\texttt{f}\ e_2\ \ldots\ (\texttt{f}\ e_n\ a))$$

$$\texttt{fold\_left f}\ a\ \texttt{[}e_1\texttt{;}e_2\texttt{;}\ldots\texttt{;}e_n\texttt{]} \;=\; (\texttt{f}\ \ldots\ (\texttt{f}\ (\texttt{f}\ a\ e_1)\ e_2)\ \ldots\ e_n)$$

```
# let rec somme = function
      [] -> 0
  | x::l -> x + somme l;;
val somme : int list -> int = <fun>
```

La fonction somme peut s'écrire

```
# let add x y = x+y;;
val add : int -> int -> int = <fun>
# let somme l = List.fold_right add l 0;;
val somme : int list -> int = <fun>
# somme [3;1;9];;
- : int = 13
```

```
# List.fold_right;;
- : f:('a ->'b ->'b) ->'a list->init:'b ->'b = <fun>
# List.fold_left;;
- : f:('a ->'b ->'a) ->init:'a ->'b list ->'a = <fun>
```

L'écriture directe de `fold_right` peut se faire de la façon suivante.

```
# let rec fold_right f l a =
    match l with
    [] -> a
 |  x::reste -> f x (fold_right f reste a);;
val fold_right:('a ->'b ->'b)->'a list ->'b ->'b =<fun>
```

On peut également redéfinir la fonction `map`.

```
# let map f l =
 List.fold_right (fun x m -> (f x)::m) l [];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

De même, la fonction qui concatène deux listes peut s'écrire :

```
# let append  l m  =
  List.fold_right (fun x n -> x::n) l m;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5;6;7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

Elle est prédéfinie dans la bibliothèque `List`.

```
# List.append [1;2;3] [4;5;6;7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```