Listes et Tableaux

I. Listes

1. Types somme

1.1. Déclaration

Un *type somme* est défini par une liste de *constructeurs* (qui commençent par une majuscule, dont l'usage est réglementé en OCaml), énumérés en utilisant « | » comme séparateurs. Par exemple,

```
type direction = Nord | Sud | Est | Ouest
```

On peut ajouter à ces constructeurs des données supplémentaires à l'aide du mot-clé `of`. Par exemple,

```
type infinint = Fin of int | Inf
type int_or_string = Int of int | String of string
type forme =
| Carre of float
| Rectangle of float * float
| Cercle of float
```

Le type que l'on définit peut dépendre d'autres types en paramètres. Voici par exemple la définition du type `option` qui est défini en OCaml :

```
type 'a option = Some of 'a | None
```

1.2. Utilisation

Il est très simple de définir un élément de type somme.

```
let a = Nord ;;
let b = Fin 12 ;;
let c = Rectangle (1., 2.) ;;
let d = Some Inf ;;
```

Pour écrire des fonctions utilisant les types sommes, on utilise la construction

```
match ... with
```

Par exemple,

```
let surface f =
    match f with
    | Carre c -> c *. c
    | Rectangle (h, l) -> h *. l
    | Cercle r -> 3.1415926535 *. r *. r
```

On appelle ce type d'opération du *filtrage* (ou *pattern matching*). Notons que le *motif* (ce qui est à la gauche de la flèche) donne la *forme* d'un terme et on nomme les variables dont on veut récupérer la valeur.

On peut utiliser « _ » comme jocker.

```
let est_carre f =
    match f with
    | Carre _ -> true
    | _ -> false
```

Exercice 1 Écrire une fonction

```
addition : infinint -> infinit -> infinit.
```

2. Listes en OCaml

On pourrait définir le type *liste* de la façon suivante : une liste, c'est soit la liste vide, soit une liste non vide constituée d'un élément et d'un liste. Cela se traduit par le type somme suivant :

```
type 'a liste = Vide | Cons of 'a * 'a liste
```

En fait, il s'agit d'un type tellement commun en OCaml (et, plus généralement, en programmation fonctionnelle) que l'on a des notations spéciales :

- ⋆ la liste vide est notée `[]` ;
- ⋆ sinon, le *cons* d'un élément e suivi d'une liste l est noté e`::`l.

De plus, on peut noter une liste dans son intégralité entre crochets, les différents éléments étant séparés par des `;`. Ainsi, `[1; 2; 3]` correspond à la liste que l'on peut aussi noter `1 :: 2 :: 3 :: []` (qui correspond à `1 :: (2 :: (3 :: []))`, on associe à droite).

Voici un exemple de fonction sur une liste :

```
let rec longueur l =
    match l with
    | [] -> 0
    | a :: l' -> 1 + longueur l'
;;
```

Elle est déjà définie, comme `List.length`, mais sa structure est typique des fonctions sur les listes basées sur un unique parcours.

Voici un exemple de fonction qui renvoit le deuxième élément d'une liste, et génère une erreur s'il n'est pas défini.

```
let second l =
    match l with
    | _ :: a :: _ -> a
    | _ -> failwith "second"
;;
```

Exercice 2 Écrire les fonctions suivantes :

1. `est_present: 'a -> 'a list -> bool`
2. `ieme: 'a list -> int -> 'a option`
3. `maximum: 'a list -> 'a`
4. `concat: 'a list -> 'a list -> 'a list`
5. `miroir: 'a list -> 'a list`

II. Programmation impérative et tableaux

1. Premiers éléments

Type `unit`

Nous allons beaucoup utiliser le type `unit` qui correspond à l'absence de valeur. C'est très utile pour des fonctions qui ont un effet, mais ne renvoient pas de résultat. Par exemple, la fonction `print_int` a pour type `int -> unit`, ce qui signifie qu'elle prend en entrée un entier, et ne renvoie rien. Entre temps, elle aura affiché la valeur de l'entier passé en argument.

On peut faire suivre une expression de type `unit` par une autre expression (d'un type quelconque) en les séparant par un point-virgule.

```
# print_int ;;     (* affichage d'entier *)
- : int -> unit = <fun>
# print_newline ;; (* passage à la ligne *)
- : unit -> unit = <fun>
# print_int 12 ;;
12- : unit = ()
# print_int 12 ; "poulet" ;;
12- : string = "poulet"
# print_int 12 ; print_newline () ;;
12
- : unit = ()
```

Notons que pour un test `if ... then ... else` dont les deux expressions sont de type unit, on peut supprimer le `else` si l'on veut ne rien faire si le teste s'évalue à `false`.

```
# if 3 mod 2 = 0 then print_string "bizarre" else () ;;
- : unit = ()
# if 3 mod 2 = 0 then print_string "bizarre" ;;
- : unit = ()
```

Boucles `for`

Commençons par un premier type de boucle, que nous allons illustrer par un exemple :

```
for i = 1 to 10 do
    print_int i ;
    print_newline ()
done ;;
```

Entre le `do` et le `done`, on a une expression de type `unit`. Notons que les bornes sont toujours incluses et que l'on a obligatoirement un pas de 1. On peut aussi aller en décroissant en remplaçant le `to` par `downto`.

Références

Avant de présenter les boucles `while`, présentons les références qui permettent d'avoir des variables dont la valeur évolue au cours du temps.

```
let factorielle n =
    let f = ref 1 in
    for i = 1 to n do
        f := !f * i
    done ;
    ! f
;;
```

- ⋆ On créée une référence en écrivant `ref` suivi de sa valeur initiale (en la mettant dans un `let`, cela permet de lui donner un nom).
- ⋆ On obtient sa valeur en faisant précéder la référence par `!`.
- ⋆ On change sa valeur à l'aide de `:=`.

Boucles `while`

Pour finir, on dispose de la boucle **while** dont la syntaxe est sans surprises.

```
let compte n =
    let k = ref n
    and cnt = ref 0 in
    while !k > 0 do
        incr cnt ; (* remplace cnt := !cnt + 1 *)
        k := !k / 2
    done ;
    !cnt
;;
```

2. Tableaux

De nombreux algorithmes impératifs portent sur des tableaux, voici quelques commandes de base.

```
# let a = [| 4; 8; 15; 18; 23; 42 |] ;;
val a : int array = [|4; 8; 15; 16; 23; 42|]
# a.(3) ;;
- : int = 18
# a.(3) <- 16 ;;
- : unit = ()
# a ;;
- : int array = [|4; 8; 15; 16; 23; 42|]
# Array.length a ;;
- : int = 6
# let b = Array.make 10 true ;;
val b : bool array =
  [|true; true; true; true; true; true; true; true; true; true|]
# Array.init 5 (fun i -> 2 * i + 1) ;;
- : int array = [|1; 3; 5; 7; 9|]
```