L2/LD – TP 2 – Types sommes et listes

Ce TP doit être fini avant la prochaine séance de TP, éventuellement sur votre temps de travail personnel.

Vous devez rendre ce TP avant 23h30 la veille de votre prochain TP. Pour cela, vous devez déposer une archive .zip contenant 1 fichier source (`.ml`) sur Moodle contenant les 4 fonctions qui résolvent les exercices du ce sujet.

1 Dépot de fichier sur Moodle

Télécharger le fichier `yen.ml`, puis déposer une archive `.zip` du fichier `yen.ml` sur Moodle.

2 Les types sommes

Un type somme est formé d'une liste de cas possible pour une valeur de ce type, chaque cas comporte un nom de cas, appelé **constructeur** et une (éventuelle) valeur associée (l'argument du constructeur) qui est un type déjà connu.

type *nouveauType*	=	Tag_1	of $typeConnu * \cdots * typeConnu$
	|	Tag_2	of $typeConnu * \cdots * typeConnu$
	|		$\cdots$
	|	Tag_n	of $typeConnu * \ldots * typeConnu$

FIGURE 1 – Définition d'un type somme

Les noms des constructeurs $Tag_1, \ldots, Tag_n$ doivent être uniques (réservés à un seul type) dans tout le fichier Caml.

2.1 le type somme prix

On peut définir le type `prix` qui contient soit un prix en euros soit un prix en yens.

À taper :
```
type prix = Euro of int
          | Yen of int
;;
```
Réponse :
```
type prix = Euro of int | Yen of int
```

On peut maintenant définir une variable de type `prix`.

À taper :
```
let p1 = Euro 3;;
```
Réponse :
```
val p1 : prix = Euro 3
```
À taper :
```
let p2 = Yen 5 ;;
```
Réponse :
```
val p2 : prix = Yen 5
```

En utilisant le filtrage, on peut écrire une fonction qui précise la monnaie utilisée :

À taper :
```
let affichemonnaie = function
      Euro x -> " le prix est en euros "
    | Yen y -> " le prix est en yen "
 ;;
```
Réponse :
```
val affichemonnaie : prix -> string = <fun>
```

À taper :
```
affichemonnaie p1;;
```
Réponse :
```
- : string = " le prix est en euros "
```
À taper :
```
affichemonnaie p2;;
```
Réponse :
```
- : string = " le prix est en yen "
```
À taper :
```
affichemonnaie 6;;
```
Réponse :
```
                -
Error: This expression has type int but an expression was expected of type
       prix
```

Exercice : Écrire une fonction `prixEnYens` qui indique la valeur en yen d'un prix (on suppose que 1 Euro = 157 Yens).

2.2 le type somme couleur

Les constructeurs d'un type somme peuvent être des constantes. Par exemple, définissons le type `couleur` comme étant {`Bleu`, `Vert`, `Rouge`}.

À taper :
```
type couleur = Bleu | Vert | Rouge;;
```
Réponse :
```
type couleur = Bleu | Vert | Rouge
```

À taper :
```
let couleurBleue = Bleu;;
```
Réponse :
```
val couleurBleue : couleur = Bleu
```
À taper :
```
let couleurVerte = Vert;;
```
Réponse :
```
val couleurVerte : couleur = Vert
```

Exercice : Écrire une fonction `estCeDuVertprix` qui indique si une couleur est du vert.

2.3 le type somme paire

On considère le type `paire` suivant :

À taper :
```
type paire = MkPaire of int * int;;
```
Réponse :
```
type paire = MkPaire of int * int
```
À taper :
```
let unePaire = MkPaire (3,2);;
```
Réponse :
```
val unePaire : paire = MkPaire (3, 2)
```

En utilisant le filtrage, on peut retrouver les valeurs contenues dans une paire :

À taper :
```
let first = function MkPaire (x,y) -> x;;
```
Réponse :
```
val first : paire -> int = <fun>
```
À taper :
```
let second = function MkPaire (x,y) -> y;;
```
Réponse :
```
val second : Paire -> int = <fun>
```

À taper : `first  (MkPaire (10,34));;`
Réponse : `- : int = 10`
À taper : `second (MkPaire (10,34));;`
Réponse : `- : int = 34`
À taper : `first unePaire;;`
Réponse : `- : int = 3`
À taper : `second unePaire;;`
Réponse : `- : int = 2`

Exercice : Écrire une fonction `max` qui retourne le maximum d'une paire d'entiers.

3 Listes en Caml

3.1 Présentation

En Caml, pour stocker plusieurs valeurs de même type, on utilise une liste :

À taper : `let l = [1;2;3;4];;`
Réponse : `val l : int list = [1; 2; 3; 4]`

`l` est une liste d'entier (`int list`). On énumère les éléments de `l` en les écrivant entre crochets, séparés par des points virgules.

À taper : `let l1 = [1.1;1.2;1.3;1.4];;`
Réponse : `val l1 : float list = [1.1; 1.2; 1.3; 1.4]`
À taper : `let l2 = ["tout";"va";"bien"];;`
Réponse : `val l2 : string list = ["tout"; "va"; "bien"]`

`l1` est une liste de réels et `l2` est une liste de chaînes de caractères.

La liste vide qui ne contient aucune valeur se note `[]`.

À taper : `let listevide = []`
Réponse : `val listevide : 'a list = []`

On remarque qu'étant donné que la liste vide ne contient aucun élément, on ne peut pas connaître le type de ses éléments donc on indique qu'ils sont du type indéfini `'a` .

La fonction suivante teste si une liste est vide :

À taper :
```
let estvide = function
          l -> (l=[])
;;
```
Réponse : `val estvide : 'a list -> bool = <fun>`

À taper : `estvide listevide;;`
Réponse : `- : bool = true`
À taper : `estvide l1;;`
Réponse : `- : bool = false`

3.2 Fonctions prédéfinies sur les listes

On peut rajouter un élément en tête de liste en utilisant la fonction `::`

À taper : `let l3 = "jusquici" :: l2`
Réponse : *`val l3 : string list = ["jusquici"; "tout"; "va"; "bien"]`*

On peut ajouter à la liste vide un élément de n'importe quel type :

À taper : `let l4 = 4.5 :: listevide`
Réponse : *`val l4 : float list = [4.5]`*
À taper : `let l5 = 3 :: listevide`
Réponse : *`val l5 : int list = [3]`*

L'élément à ajouter doit être du même type que les éléments de la liste :

À taper :
```
4.5 :: l5;;
```
Réponse :
```
        --
  Error: This expression has type int list
     but an expression was expected of type float list
```

Pour créer une liste à partir de deux listes, on utilise la fonction `@` appelée concaténation :

À taper : `[1]@[1;2;3];;`
Réponse : *`- : int list = [1; 1; 2; 3]`*
À taper : `1@[1;2;3];;`
Réponse :
```
-
Error: This expression has type int but an expression was expected of type 'a list
```

3.3 fonctions récursives

Lorsque l'on utilise les listes, on définit souvent des fonctions récursives :

À taper :
```
let rec somme = function
    [] -> 0
  | x::l -> x + somme l
```
Réponse : *`val somme : int list -> int = <fun>`*
À taper : `somme l;;`
Réponse : *`- : int = 10`*

La fonction `somme` fait la somme des éléments de la liste passée en arguement. Elle est définie par filtrage : soit la liste est vide, soit elle est composée d'un premier élément x suivi de la fin de la liste.

Exercice : Écrire une fonction `max` qui retourne le nombre d'éléments non nuls contenus dans une liste.