TD de programmation fonctionnelle et logique
Corrigé du TD 1 : définitions, filtrage

1 Définitions

1. Définissez une variable `a` égale à `3*b` où `b` vaut 7. Attention, nous ne voulons pas que `b` soit définie globalement !
```
# let a = let b = 7 in 3*b;;
val a : int = 21
```
2. Définissez une fonction qui à l'entier `x` associe `x+a` où `a` vaut localement 3.
```
# let f x = let a = 3 in x+a;;
val f : int -> int = <fun>
```
3. Définissez une fonction `f` qui à l'entier `x` associe l'entier `2*x+1`.
```
# let f x = 2*x+1;;
val f : int -> int = <fun>
```
4. Définissez une fonction `g` qui à l'entier `x` associe $(2*x+1)^2$.
```
# let g x = (2*x+1)*(2*x+1);;
val g : int -> int = <fun>
```
5. Même question que précédemment, mais en ne calculant qu'une fois la valeur de `2*x+1` (et sans utiliser de fonction "carré").
```
# let g x = let a = 2*x+1 in a*a;;
val g : int -> int = <fun>
```
6. Même problème qu'à la question 4, mais en utilisant une fonction locale pour calculer le produit `2*x+1`.
```
# let g x = let f y = 2*y+1 in (f x)*(f x);;
val g : int -> int = <fun>
```
7. Même problème qu'à la question précédente, mais en n'effectuant qu'un seul appel à la fonction locale.
```
# let g x = let f y = 2*y+1 in let a = f x in a*a;;
val g : int -> int = <fun>

# let g x = let a = (let f y = 2*y+1 in f x) in a*a;;
val g : int -> int = <fun>
```

2 Filtrage

Attention : tous les filtrages doivent être exhaustifs !

1. Nous voulons trois versions d'une fonction qui prend une paire d'entiers en argument et qui renvoie la somme des éléments de la paire :
   (a) une version simple (sans filtrage) ;
   (b) une version avec filtrage par fonction anonyme ;
   (c) une version avec filtrage explicite (et prenant en entrée un unique argument `p`).

```
# let sum (a,b) = a+b;;
val sum : int * int -> int = <fun>

# let sum = function
    (a,b) -> a + b;;
val sum : int * int -> int = <fun>

# let sum p = match p with
    (a,b) -> a + b;;
val sum : int * int -> int = <fun>
```

2. Écrivez une fonction qui prend en entrée une liste et qui renvoie la somme des deux premiers entiers la constituant. Proposez plusieurs solutions (avec et sans filtrage (explicite ou non)).

```
# let somme_deux (a::b::r) = a+b;;
# let somme_deux (a::b::r) = a+b;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val somme_deux : int list -> int = <fun>

# let somme_deux = function
      []    -> failwith "pas assez d'arguments"
  |   [a]   -> failwith "pas assez d'arguments"
  | a::b::r -> a+b;;
val somme_deux : int list -> int = <fun>

# let somme_deux = function
    a::b::r -> a+b
  |    _    -> failwith "pas assez d'arguments";;
val somme_deux : int list -> int = <fun>

# let somme_deux l = match l with
    a::b::r -> a+b
  |    _    -> failwith "pas assez d'arguments";;
val somme_deux : int list -> int = <fun>
```

3. Écrivez deux versions d'une fonction qui prend en entrée les coordonnées de deux points (sous forme de paires de réels) et qui renvoie le carré de la longueur du segment correspondant : une version simple et une avec un filtrage explicite (la fonction prend alors en entrée deux arguments `a` et `b`).

```
# let longueur (a,b) (c,d) = (a-.c)*.(a-.c) +. (b-.d)*.(b-.d);;
val longueur : float * float -> float * float -> float = <fun>

# let longueur a b =
  match (a,b) with ((x,y),(z,t)) -> (x-.z)*.(x-.z) +. (y-.t)*.(y-.t);;
val longueur : float * float -> float * float -> float = <fun>
```

4. Écrivez une fonction qui prend en entrée une liste de paires et qui renvoie la somme des éléments de son premier élément (sic).

```
# let liste_de_paires = function
    (a,b)::_ -> a+b
  |   _      -> failwith "liste vide";;
val liste_de_paires : (int * int) list -> int = <fun>
```

Exemple d'utilisation :

```
# liste_de_paires [(3,4);(5,6);(7,8)];;
- : int = 7
```

5. Écrivez une fonction qui prend en argument une paire de listes et qui renvoie la somme du premier élément de la première liste et du premier élément de la deuxième.

```
# let paire_de_listes = function
    (a::_,b::_) -> a+b
  |      _      -> failwith "une des deux listes est vide";;
val paire_de_listes : int list * int list -> int = <fun>
```

6. Même question que précédemment, mais sans utiliser le symbole « _ ».

```
# let paires_de_listes = function
     ([],[])   -> failwith "les deux listes sont vides"
  |  ([],t::r) -> failwith "la première liste est vide"
  | (t::r,[])  -> failwith "la deuxième liste est vide"
  | (a::b,c::d) -> a+c;;
val paires_de_listes : int list * int list -> int = <fun>
```

7. Écrivez une fonction qui prend en entrée une paire et une liste et qui fait la somme des premiers éléments de ses deux arguments.

```
# let paire_et_liste p l = match (p,l) with
    ((a,_),t::_) -> a+t
  |       _      -> failwith "Le deuxième argument est une liste vide";;
val paire_et_liste : int * 'a -> int list -> int = <fun>
```

Exemple d'utilisation :

```
# paire_et_liste (1,3) [7;12;45];;
- : int = 8
```

8. Écrivez une fonction qui prend en entrée une liste de trois entiers, notée `[a;b;c]` et qui renvoie la liste `[c;b;a;b;c]`. Cette fonction devra utiliser un synonyme.

```
# let liste_avec_synonyme = function
    [a;b;c] as l -> c::b::l
  |     _        -> failwith "la liste doit avoir exactement trois arguments";;
val liste_avec_synonyme : 'a list -> 'a list = <fun>
```