TD 3 Programmation fonctionnelle

Écrire les fonctions qui prennent en entrée une liste et calculent :

1. La somme des éléments de la liste.

```
sum [1; 2; 3] --> 6
```

Solution.

```
let rec sum ls =
 match ls with
 | [] -> 0
 | hd::tl -> hd + sum tl
```

□

2. Si la liste contient zéro.

```
contains_zero [12; 34; 0; 15] --> true
```

Solution.

```
let rec contains_zero ls =
  match ls with
  | [] -> false
  | 0::tl -> true
  | hd::tl -> contains_zero tl
```

□

3. Si la liste contient le nombre spécifié.

```
contains 81 [77; 81; 15; 82] --> true
```

Solution.

```
let rec contains x ls =
 match ls with
 | [] -> false
 | hd::tl -> (hd = x) || contains x tl
```

□

4. Si la liste est triée en ordre croissant.

```
is_sorted [5; 10; 15; 21] --> true
```

Solution.

```
let rec is_sorted ls =
 match ls with
 | [] -> true
 | a :: [] -> true
 | a :: b :: tl -> a <= b && is_sorted (b::tl)
```

Variant

```
let rec is_sorted ls =
 match ls with
 | a :: b :: tl -> a <= b && is_sorted (b::tl)
 | _ -> true    (* Wildcard pattern _ matches anything *)
```

□

5. Si la liste contient deux éléments identiques consécutifs.

   `contains_pair [10; 7; 15; 15; 28]--> true`

Solution.

```
let rec contains_pair ls =
 match ls with
 | a :: b :: tl -> (a = b) || contains_pair (b::tl)
 | _ -> false
```

□

6. La liste où tous les éléments sont incrémentés de 1.

   `increment_all [1; 2; 3; 4; 5] --> [2; 3; 4; 5; 6]`

Solution.

```
let rec increment_all ls =
 match ls with
 | [] -> []
 | hd :: tl -> (hd + 1) :: increment_all tl
```

□

7. La liste de parité des entiers (pairs ou impairs).

   `parité [1 ; 2 ; 3 ; 4 ; 5] --> [true ; false ; true ; false ; true]`

Solution.

```
let rec parity ls =
 match ls with
 | [] -> []
 | hd :: tl -> (hd mod 2 <> 0) :: parity tl
```

□

8. La liste qui conserve uniquement les nombres pairs

```
keep_even [1 ; 2 ; 3 ; 4 ; 5] --> [2 ; 4]
```

Solution.

```
let rec keep_even ls =
 match ls with
 | [] -> []
 | hd :: tl ->
     if hd mod 2 = 0 then
       hd :: keep_even tl
     else
       keep_even tl
```

□

9. La liste qui insère après

```
insert_after 100 [1 ; 2 ; 3] --> [1 ; 100 ; 2 ; 100 ; 3 ; 100]
```

Solution.

```
let rec insert_after v ls =
 match ls with
 | [] -> []
 | hd::tl -> hd :: v :: insert_after v tl
```

□

10. ou entre

```
insert_between 100 [1 ; 2 ; 3] --> [1 ; 100 ; 2 ; 100 ; 3]
```

Solution.

```
let rec insert_between v ls =
 match ls with
 | [] -> []
 | a :: [] -> [a]
 | a :: tl -> a :: v :: insert_between v tl
```

Variant

```
let rec insert_between v ls =
  match ls with
  | a :: b :: tl -> a :: v :: insert_between v (b::tl)
  | _ -> ls
```

□

11. La liste qui entrelace deux listes

    `interleave [1 ; 2 ; 3] [100 ; 200 ; 300] --> [1; 100; 2; 200; 3; 300]`

    Si l'une des listes est plus courte :

    `interleave [1; 2] [100; 200; 300; 400] --> [1; 100; 2; 200; 300; 400]`

    `interleave [1; 2; 3; 4] [100; 200] --> [1; 100; 2; 200; 3; 4]`

Solution.

```
let rec interleave ls1 ls2 =
 match ls1 with
 | hd :: tl -> hd :: interleave ls2 tl
 | [] -> ls2
```

□

12. l'aplatissement de la liste

    `flatten [[1;2;3]; [8;9]; []; [4; 5]] --> [1; 2; 3; 8; 9; 4; 5]`

Solution.

```
let rec flatten ls =
 match ls with
 | [] -> []
 | [] :: tl -> flatten tl
 | (x::xs) :: tl -> x :: flatten (xs :: tl)
```

□