DM1 : Récursivité, Listes

18 septembre 2023

Ce DM est à rendre le **2 octobre 2023**. Le rendu se fera sur copie. Il est fortement conseillé de tester vos fonctions sur machine sur quelques exemples.

Le DM est composé de trois exercices et d'un problème indépendants.

Exercice 1. Fonctions de bases

1. Proposer une fonction OCaml `f` de signature `int -> int` telle que `f n` ait pour valeur $\frac{n(n-1)}{2}$

```
f 4 (* 6 *)
```

2. Proposer une fonction `plus_grand` à deux arguments qui renvoie le plus grand de ses deux arguments.

```
plus_grand 1 2 (* 2 *)
```

```
plus_grand 8 3 (* 8 *)
```

3. Proposer une fonction OCaml `est_positif` de signature `int -> bool` de sorte à ce que `est_positif x` soit égal à `true` si et seulement si $x \geqslant 0$.

```
est_positif (-2) (* false *)
```

```
est_positif 4 (* true *)
```

Correction :

1.
```
let f n = n * (n-1) / 2
```

2. On a plusieurs possiblités, on peut utiliser un filtrage ou une structure conditionnelle :

```
let plus_grand x y = match x>y with
| true -> x
| _ -> y
```

```
let plus_grand x y = if x>y then x else y
```

3.
```
let est_positif x = x >= 0
```

Exercice 2. Suite récurrente et récursivité

Pour chacune des fonctions suivantes, proposer une implémentation récursive pour les fonctions suivantes :

1. Proposer une fonction OCaml `u` de type `int -> int` de sorte à ce que `u n` calcule le terme u_n défini par :

$$u_0 = 0$$
$$\forall n > 0, u_n = u_{n-1} + 3$$

2. Proposer une fonction OCaml `v` de type `int -> int` de sorte à ce que `v n` calcule le terme v_n défini par :

$$v_0 = 1$$
$$\forall n > 0, v_n = 2v_{n-1}$$

3. Proposer une fonction OCaml `w` de type `int -> int -> int` de sorte à ce que `w a n` calcule le terme w_n défini par :

$$w_0 = 0$$
$$\forall n > 0, w_n = w_{n-1} + a$$

4. Proposer une fonction OCaml `suite_geometrique` de type `int -> int -> int -> int` de sorte à ce que `suite_geometrique a q n` calcule le terme z_n défini par

$$z_0 = a$$
$$\forall n > 0, z_n = qz_{n-1}$$

Correction :

1.
```
let rec u n = match n with
| 0 -> 0
| _ -> 3 + u (n-1)
```

2.
```
let rec v n = match n with
| 0 -> 1
| _ -> 2 * v (n-1)
```

3.
```
let rec w a n = match n with
| 0 -> 0
| _ -> a + w a (n-1)
```

4.
```
let rec suite_geometrique a q n = match n with
| 0 -> a
| _ -> q * suite_geometrique a q (n-1)
```

Exercice 3. Listes

1. Proposer une fonction OCaml `nombre_elements_positifs` de signature `int list -> int` telle que `nombre_elements_positifs l` renvoie le nombre d'éléments positifs dans `l`.

```
nombre_elements_positifs [0; 2; -1; 4] (* 3 *)
```

2. Proposer une fonction OCaml `filtrer_positifs` de signature `int list -> int list` telle que `filtrer_positifs l` renvoie une liste des éléments positifs de `l`.

```
filtrer_positifs [0; 2; -1; 4] (* [0; 2; 4]*)
```

3. Proposer une fonction OCaml `dernier_positif` de signature `int list -> int` telle que `dernier_positif l` renvoie le dernier élément positif de `l`, et 0 à défaut.

```
dernier_positif [2; -1; 3; -2] (* 3 *)
```

```
dernier_positif [-1] (* 0 *)
```

```
dernier_positif [4; 1; 0; -2] (* 0 *)
```

Correction :

1.
```
let rec nombre_elements_positifs l = match l with
| [] -> 0
| p::q when p>=0 -> 1 + nombre_elements_positifs q
| _::q -> nombre_elements_positifs q
```

2.
```
let rec filtrer_positifs l = match l with
| [] -> 0
| p::q when p>=0 -> p::filtrer_positifs q
| _::q -> filtrer_positifs q
```

3. On utilise une fonction auxilaire qui a un argument supplémentaire dans lequel on met le résultat.

```
let dernier_positif l=
    let rec aux l acc = match l with
    | [] -> acc
    | p::q when p>= 0 -> aux q p
    | _::q -> aux q acc
    in aux l 0
```

Problème : Fonction d'itération

On cherche à implémenter une fonction qui permette d'itérer une autre fonction passée en argument de sorte à pouvoir émuler le comportement python suivant :

```
def iterer(f, n, x)
    """Calcule f(f(....f(x)....) avec n applications de f"""
    for _ in range(n):
        x = f(x)
    return x
```

1. Proposer une fonction OCaml `iterateur` de sorte à ce que, pour toute fonction `f` et toute variable `x`, `iterateur f n x` soit égal à `f (f (.... (f x) ... ))` avec `f` itérée `n` fois.

```
iterateur (fun x -> x * 2) 5 1 (*32*)
```

On pourra chercher une relation de récurrence entre `iterateur f n x` et `iterateur f (n-1) x`.

2. Quelle est la signature de la fonction `iterateur` ?
3. En déduire une implémentation de la fonction puissance de deux qui corresponde à $f(n) = 2^n$.

4. On cherche désormais à pouvoir utiliser la valeur du compteur actuel pour pouvoir obtenir le comportement python suivant :

```
def iterer(f, n, x)
    """Calcule f(f(....f(x)....)) avec n applications de f"""
    for k in range(n):
        x = f(k, x)
    return x
```

Proposer une implémentation d'une fonction `iterateur2` de sorte à ce que pour toute fonction `f`, `iterateur f n x` soit égal à `f (n−1) (f (n−2) (f ... (f 0 x) ...))`

5. Quelle est la signature de la fonction `iterateur2` ?
6. En déduire une implémentation permettant de calculer $f(n) = \sum_{k=0}^{n} k^2$.

Correction :

1. On a deux possibilités, soit on utilise le fait que $f^{(n)}(x) = f(f^{(n-1)}(x))$, soit on utilise le fait que $f^{(n)}(x) = f^{(n-1)}(f(x))$.

```
let rec iterateur f n x = match n with
| 0 -> x
| _ -> f (iterateur f (n-1) x)
```

```
let rec iterateur f n x = match n with
| 0 -> x
| _ -> iterateur f (n-1) (f x)
```

La deuxième méthode a l'avantage d'être récursive terminale.

2. La signature est `('a -> 'a) -> int -> 'a -> 'a`.
3. On réutilise la fonction iterateur :

```
let puissance2 n = iterateur (fun x -> 2 * x) n 1
```

```
let rec iterateur2 f n x = match n with
| 0 -> x
| _ -> f (n-1) (iterateur2 f (n-1) x)
```

5. La signature de `iterateur2` est `(int -> 'a -> 'a) -> int -> 'a -> 'a`.
6. On utilise la fonction `iterateur2` :

```
let somme_carres n = iterateur2 (fun n x -> x + n*n) (n+1) 0
```