TP3 : Programmation Impérative en OCaml

Semaine du 25 Septembre 2023

Exercice 1. Conditions

À l'aide de la syntaxe `if ... then ... else ...`, proposer une implémentation des fonctions suivantes :

1. Une fonction de signature `int -> bool` qui détermine si une année est bissextile.
2. Une fonction de signature `'a -> 'a -> 'a` qui renvoie l'élément le plus grand parmi les deux arguments.
3. Une fonction de signature `bool -> bool -> bool` qui renvoie `true` si et seulement si un nombre pair de ses deux entrées est égal à `true`.
4. Une fonction de signature `bool -> bool-> bool -> bool` qui renvoie `true` si et seulement si un nombre pair de ses arguments est égal à `true`.

   On veillera à parenthéser correctement les conditions pour éviter des problèmes d'associativité.
5. Une fonction `clamp` de signature `float -> float -> float ->float` telle que, pour $a < b$, `clamp a b x` soit égal à $f_{a,b}(x)$ défini de la manière suivante :

$$f_{a,b}(x) = \begin{cases} b \text{ si } x > b, \\ a \text{ si } a > x, \\ x \text{ sinon.} \end{cases}$$

Exercice 2. Référence

Proposer une implémentaion des fonctions suivantes :

1. Une fonction `raz` de signature `int ref -> unit` qui assigne 0 à la référence passée en argument.
2. Une fonction `incrementer` de signature `int ref -> unit` qui augmente de 1 la référence passée en argument.
3. Une fonction `tester_et_assigner` de signature `bool ref -> bool` qui assigne `true` à une référence passé en argument et renvoie l'ancienne valeur dans la référence.
4. Une fonction `assigner_et_renvoyer` de signature `'a ref -> 'a ->'a` de sorte à ce que `assigner_et_renvoyer r x` assigne `x` à la référence `r` et renvoie l'ancienne valeur de `r`.
5. Une fonction `echanger` de signature `'a ref -> 'a ref -> unit` qui échange la valeur contenue dans deux références.

Exercice 3. Boucles

Proposer une implémentation impérative des fonctions suivantes :

1. Une fonction de signature `int -> int` qui calcule 2^n où n est son entrée.
2. Une fonction de signature `int -> int` qui calcule $n!$ où n est son entrée.
3. Une fonction de signature `int -> int` qui sur une entrée n calcule la somme des entiers jusqu'à n, $\sum_{k=0}^{n} k$.
4. Une fonction de signature `int -> float` qui calcule la suite définie par récurrence suivante :

$$v_0 = 2$$
$$\forall n \geqslant 0, v_{n+1} = \frac{1}{2}\left(v_n + \frac{2}{v_n}\right)$$

   Attention à ne calculer qu'une seul fois v_n.
5. Une fonction de signature `int -> int` qui renvoie le plus petit $n \geqslant 0$ tel que 2^n soit plus grand que son entrée.
6. Une fonction de signature `int -> int -> int` qui calcule le PGCD de ses deux arguments avec l' algorithme d'Euclide.

   Attention : vous ne pouvez pas utiliser un nom pour une variable ou une fonction qui commence par une majuscule.
7. Une fonction de signature `int -> int` qui renvoie le nombre de diviseurs de son entrée.
8. Une fonction de signature `int -> int` qui calcule le n-ième terme de la suite de fibonacci définie par

$$u_0 = 0$$
$$u_1 = 1$$
$$\forall n \in \mathbb{N}\ u_{n+2} = u_{n+1} + u_n$$

   On veillera à ne pas trop faire d'appels récursifs.

Exercice 4. Traitement impératif des listes

L'une des forces d'OCaml est de mêler aspects impératifs et fonctionnels : il est possible de traiter un problème à la fois de manière impérative et fonctionelle de manière cohérente.

En utilisant des références de liste, proposer des solutions aux questions suivantes :

1. Proposer une fonction de signature `'a list -> bool` qui renvoie si la liste en entrée est vide.
2. Proposer une fonction de signature `int list -> int` qui renvoie la somme des éléments d'une liste.
3. Proposer une fonction de signature `'a list -> bool` qui renvoie `true` si et seulement si la liste en entrée a un nombre pair d'éléments.
4. Proposer une fonction de signature `float list -> float` qui renvoie le produit des éléments d'une liste.
5. Proposer une fonction de signature `int list -> int` qui renvoie le maximum des éléments d'une liste.

   On pourra renvoyer `min_int` si la liste est vide.
6. Proposer une fonction de signature `int list -> int` qui renvoie le minimum des éléments d'une liste.
7. Proposer une fonction `dernier_element` de signature `'a list -> 'a` qui renvoie le dernier élément de la liste en entrée.
8. Proposer une fonction de signature `int list -> int liste` qui renvoie une liste de tous les éléments non nuls de la liste en entrée.

9. Proposer une fonction concat de signature 'a list -> 'a list -> 'a list qui renvoie la concaténation des éléments des deux listes en entrée.

```
concat [1;2] [3;4;5] (* [1;2;3;4;5] *)
```

OCaml propose l'opérateur @ qui fait la même chose.

10. Proposer une fonction inverse de signature 'a list -> 'a list de sorte à ce que inverse l renvoie la liste l inversée.

```
inverse [1;2;3] (* [3;2;1] *)
```

11. Proposer une fonction inserer de signature 'a list -> int -> int -> 'a list de sorte à ce que inserer l n x renvoie la liste l mais avec l'élément x inséré à la position n.

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

Dans le cas où la liste est trop petite, on pourra lever une exception, ou ajouter l'élément à la fin de la liste.

12. Proposer une fonction zip de signature 'a list -> b' list -> ('a * 'b) list de sorte à ce que, pour deux listes de même longueurs [a1; ...; an] et [b1 ;...; bn], zip [a1; ...; an] [b1; ...; bn] renvoie la liste de couple [(a1,b1) ;... ; (an; bn)].
13. Proposer une fonction unzip de signature ('a * 'b) list -> 'a list * b' list qui sur une entrée [(a1,b1) ;... ; (an; bn)] renvoie le couple de listes ([a1; ...; an], [b1; ...; bn]).
14. Proposer une fonction mirroir de type ('a*'b) list -> ('b*'a) list qui inverse l'ordre de tous les couples de la liste en argument :

```
mirroir [(1,2);(3,4)] (* [(2,1);(4,3)] *)
```

15. Proposer une fonction pairs_avec de type 'a -> 'b list -> ('a*'b) list qui réalise une liste des couples dont le premier élément est le premier argument, et dont les deuxièmes éléments sont les éléments de la liste en argument.

```
pairs_avec 1 [2;3;4] (*[(1,2); (1, 3); (1,4)]*)
```

16. Proposer une fonction pairs de signature 'a list -> ('a * 'a) list qui renvoie la liste de toutes les pairs d'éléments de la liste en entrée.

```
pairs [1;2;3] (* [(1,2); (1,3); (2,3); (2, 1); (3, 1); (3,2)]*)
```

L'ordre n'est pas important. On pourra utiliser l'exercice 9.

17. Proposer une fonction ajouter_a_toutes de signature 'a -> 'a list list -> 'a list list qui, à partir d'un élément et d'une liste de listes, rajoute l'élément dans toutes les listes de la liste.

```
ajouter_a_toutes 1 [[];[4];[2;3]] (* [[1];[1;4];[1;2;3]]*)
```

18. Proposer une fonction produit_cartesien de signature 'a list -> 'b list -> ('a * 'b) list qui renvoie la liste des couples dont le premier élément est dans la liste en premier argument, et dont le second élément est dans la liste en second argument.

```
produit_cartesien [1;2] [3;4;5] (*[(1,3); (1,4); (1,5); (2,3); (2,4); (2,5)
```

19. Proposer une fonction prefixes de signature 'a list -> 'a list list qui renvoie la liste de tous les préfixes de la liste en entrée.

```
prefixes [1;2;3] (* [[];[1];[1;2];[1;2;3]]*)
```

20. Proposer une fonction sous_ensembles de signature 'a list -> 'a list list qui renvoie la liste de toutes les sous-liste incluses dans l'entrée, en gardant l'ordre des éléments de l'entrée.

```
sous_ensembles [1;2;3] (*[[]; [1]; [2]; [3]; [1; 2]; [2; 3]; [1; 3]; [1; 2; 3]]*)
```

Les structures récursives comme les listes sont généralement plus facile à manipuler avec des fonctions récursives, mais nous verrons des cas où un traitement impératif peut être plus pratique.

Exercice 5. Affichage

1. Sans utiliser `Printf.printf` proposer une fonction de signature `string -> int -> string -> unit` qui affiche l'entier passé en argument entre les deux chaînes passées en argument.
2. Proposer une fonction de signature `int list -> unit` qui affiche le contenu d'une liste passée en argument. On prendra soin d'afficher des délimiteurs entre les éléments de la liste, ainsi que des crochets sur les extrémités.
3. Proposer une fonction de signature `int list list -> unit` qui affiche le contenu d'une liste de listes d'entiers passée en argument.

Exercice 6. Variable de retour dans une référence

Dans certaines circonstances, on peut être intéressé par stocker le résultat d'une fonction dans une référence passée en argument au lieu de simplement renvoyer le résulat.

```
let f x r = r:=x*2
```

1. Quel est le type de la fonction `f` ?
2. Proposer une implémentation de la fonction factorielle qui stocke le résultat dans une référence passée en argument.
3. Proposer une fonction `sauvegarder` de signature `('a -> 'b) -> 'a -> 'b ref -> unit` qui prend en argument une fonction, un argument et une référence et qui stocke le résultat de la fonction sur l'argument dans la référence.

Cette astuce permet d'avoir plusieurs variables de retour dans certains langages.

Exercice 7. Un peu de mémoire

On cherche à calculer la suite définie par récurrence suivante :

$$u_0 = 0.25$$
$$\forall n > 0, u_n = u_{n-1}^2 + \frac{u_{n-1}}{2}$$

On cherche a utiliser des références pour se souvenir de la plus grande valeur de u_n qu'on ait calculée pour pouvoir repartir depuis cette valeur là au lieu de repartir depuis zéro quand on cherche à calculer une valeur plus grande.

On se donne a deux références globales pour enregistrer le n de la dernière valeur de u_n qu'on a calculée, ainsi que la valeur associée.

```
let n_max = ref 0
let u_max = ref 0.25
```

Notre méthode est la suivante : à chaque fois qu'on doit calculer u_n pour un n donné, on vérifie si $n \geqslant n_{\max}$. Si c'est le cas, on repars depuis les valeurs stockées dans les références, sinon, on repars depuis zéro.

Dans les deux cas, on met à jour `n_max` et `u_max` pour une utilisation future.

1. Proposer une fonction `u` de signature `int -> float` qui réalise cette implémentation.

2. Dans quels cas cette implémentation est particulièrement intéressante ?

Exercice 8. Compteur

Proposer une fonction de signature `int -> unit -> int` qui sur une entrée n renvoie une fonction qui, à chaque appel, renverra successivement 0, n, $2n$, ...

On veillera à ce que les compteurs soient indépendants.

```
let c0 = compteur 1;;
let c1 = compteur 2;;
```

```
c0 () (* 0 *)
```

```
c0 () (* 1 *)
```

```
c1 () (* 0 *)
```

```
c0 () (* 2 *)
```

```
c1 () (* 2 *)
```

On pourra chercher dans quel contexte il faut définir la nouvelle référence qui servira pour la fonction renvoyée.

Exercice 9. Boucle sans rec ni while ni for

On propose le code suivant :

```
let f n =
        let r = ref (fun x -> 0) in
            (r:= fun x -> if x = 0 then 1 else x * (!r)(x-1)) ;
            !r n
```

1. Que fait la fonction `f` ?
2. Sur le même principe, proposer une implémentation des fonctions de l'exercice 3.