# Premiers exos OCaml : Corrigé partiel

**Exercice 13.** *Des fonctionnelles simples.* Écrire une fonction (et prévoir son type) qui à deux fonctions $f$ et $g$ associe :

— la fonction composée $f \circ g$ ;

— la fonction $\min(f, g)$

— la fonction $\max(f \circ g, g \circ f)$.

**Corrigé.** On utilise la déclaration de fonctions en leur donnant un nom, plutôt que `fun` ou `function`.

1. `let compose f g x = f (g x) ;;`
   de type `('a -> 'b) -> ('c -> 'b) -> 'c -> 'b`. En effet, le type d'entrée de `f` est le même que celui de sorti de `g`.
2. La fonction `min` de Ocaml est une fonction polymorphe à deux arguments. On propose
   `let mini f g x = min (f x) (g x) ;;` de type `('a -> 'b) -> ('a -> 'b) -> 'b`. En effet les types d'entrée des deux fonctions `f` et `g` sont les mêmes, de même que les types de sortie.
3. `let maxi_compose f g x = max (f (g x)) (g (f x)) ;;`
   de type `('a -> 'a) -> ('a -> 'a) -> 'a -> 'a`. En effet, pour composer $g \circ f$ et $f \circ g$, les types d'entrée et de sortie de `f` et `g` doivent tous être identiques.

**Exercice 14.** *Reconnaissance de types fonctionnels.* Reconnaître le type des fonctions `f`, `g` et `h` suivantes :

```
let f x y = x y ;;
let g x y = x (x y) ;;
let h x y = x (y x) ;;
```

**Corrigé.**

1. Pour `f`, `x` est une fonction et `y` a le type d'entrée de `x`. D'où le type : `('a -> 'b) -> 'a -> 'b`.
2. Pour `g`, c'est pareil, mais comme on recompose avec `x`, le type de sortie de `x` est le même que son type d'entrée. D'où le type `('a -> 'a) -> 'a -> 'a`.
3. Pour `h`, `y` est maintenant une fonction prenant en entrée le type de `x` et renvoyant en sortie le type d'entrée de `x`. D'où `('a -> 'b) -> (('a -> 'b) -> 'a) -> 'b`.

**Exercice 15.** *Reconnaissance de type.* Reconnaître la signature (le type) des fonctions suivantes :

```
let f (x, y) = if x then y else 0;;
let g x y = if x > y then (x,y) else (y,x);;
let h (x,y,z) = function x -> (x,y,z);;
```

**Corrigé.**

1. Pour `f`, `x` est nécessairement de type booléen, et `y` de type entier par homogénéité dans la structure conditionnelle (type de la valeur suivant le `then` du même type que celle qui suit le `else`). D'où le type `bool * int -> int`.
2. Pour `g`, `x` et `y` sont de même type pour être comparables, d'où le type de la fonction : `'a -> 'a -> 'a * 'a`.
3. Pour `h`, attention, le `x` apparaissant après `function` est « muet » et n'est pas le même que celui apparaissant dans l'entrée de `h`, qui ne sert pas. D'où le type `'a * 'b * 'c -> 'd -> 'd * 'b * 'c`.

**Exercice 19.** Écrire une fonction `nbchiffres: int -> int` donnant le nombre de chiffres d'un entier (on pourra convenir que 0 n'a aucun chiffre).

**Corrigé.** Rappel : / est la division entière sur les entiers.

```
let rec nbchiffres n =
  let a=ref n and c = ref 0 in
  while !a>0 do
    incr c ; a:= !a / 10
  done ;
  !a
;;
```

**Exercice 22.** La suite de Fibonacci est définie par $F_0 = 0$, $F_1 = 1$, et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$. Écrire une fonction `fibo: int -> int` calculant le terme d'indice $n$ de la suite.

**Corrigé.**

```
let fibo n =
  let a=ref 0 and b=ref 1 in
  for i=2 to n do
    let c= !a + !b in a:= !b ; b:= c
  done ;
  if n=0 then !a else !b
;;
```

**Exercice 23.** *Suite de Syracuse.* La suite de Syracuse est définie par $u_0 = a \in \mathbb{N}^*$ et $u_{n+1} = \begin{cases} u_n/2 & \text{si } n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$ pour tout $n \geq 0$. Il est conjecturé que pour tout choix de $u_0$, il existe un indice $p$ de la suite tel que $u_p = 1$. Écrire une fonction `syracuse: int -> unit` affichant la suite des valeurs de $u_0 = a$ à $u_p = 1$, séparées par des espaces, avec $p$ le premier indice tel que $u_p = 1$. (Rappel : `print_int` pour afficher un entier, `print_string " "` pour afficher un espace à l'écran).

**Corrigé.**

```
let syracuse n =
  let u = ref n in
  while !u>1 do
    print_int !u ; print_string " " ;
    u:= if !u mod 2 = 1 then 3 * !u + 1 else !u/2
  done ;
  print_int 1
;;
```

**Exercice 24.** *Multiplication du paysan russe.* On peut calculer le produit $ab$ de la façon suivante : on divise $a$ par 2 tant que possible, en multipliant $b$ par deux en parallèle. Si $a$ est impair, on décrémente $a$ et on rajoute $b$ au résultat final. Le procédé s'arrête lorsque $a$ tombe à zéro. Écrire une fonction `mult_paysan: int -> int -> int` implémentant cette idée.

**Corrigé.** On utilise deux variables locales qui sont des références vers les valeurs passées en paramètres (et on leur donne le même nom !) Il est aussi nécessaire d'utiliser un accumulateur (`res`) pour stocker le résultat.

```
let mult_paysan a b=
  let res = ref 0 in
  let a=ref a and b=ref b in
  while !a>0 do
    if !a mod 2 = 0 then
      (a:= !a / 2 ; b:= 2 * !b)
    else
      (res := !res + !b ; decr a)
  done ;
  !res
;;
```

**Exercice 37.** *Tableau de Fibonacci.* Écrire une fonction `tfibo : int -> int array` renvoyant le tableau constitué des $n$ premiers termes de la suite de Fibonacci (avec $F_0 = F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$).

**Corrigé.**

```
let tab_fibo n=
  let t=Array.make n 1 in
  for i=2 to n-1 do
    t.(i) <- t.(i-1) + t.(i-2)
  done ;
  t
;;
```

**Exercice 39.** *Appliquer une même fonction à tous les éléments d'un tableau.* Écrire une fonction `mapt` de type `('a -> 'b) -> 'a array -> 'b array` qui applique une fonction `f` à tous les éléments d'un tableau `v`. (Remarque : `Array.map f t` fait exactement ça, on ne l'utilisera pas).

**Corrigé.** On suppose le tableau d'entrée non vide.

```
let mapt f t=
  let n=Array.length t in
  let t2=Array.make n (f (t.(0))) in
  for i=1 to n-1 do
    t2.(i) <- f (t.(i))
  done ;
  t2
;;
```

**Exercice 42.** *Décalage à droite et à gauche.* Écrire `decag: 'a array -> unit` et `decad: 'a array -> unit` deux fonctions effectuant un décalage d'une position vers la gauche (resp. vers la droite) dans un tableau (l'élément sortant faisant sa rentrée du coté opposé, votre fonction modifie le tableau en place).

**Corrigé.**

```
let decag t =
  let x=t.(0) and n=Array.length t in
  for i=1 to n-1 do
    t.(i-1) <- t.(i)
  done ;
  t.(n-1) <- x
;;

let decad t =
  let x=t.(n-1) and n=Array.length t in
  for i=0 to n-2 do
    t.(i+1) <- t.(i)
  done ;
  t.(0) <- x
;;
```