Chapitre 4
Correction des exercices

Exercice 1 La fonction suivante réalise la permutation du contenu de deux références polymorphes :

```
let swap x y =
  let z = !x in
    x := !y ; y := z ;;
```

La fonction `f` de l'énoncé fait de même avec des références de type entier (car on utilise addition et soustraction du type *int*). Son type est donc *int ref –> int ref –> unit*. Observons en effet l'évolution du contenu des deux références `x` et `y` lors de la séquence d'instructions qu'applique cette fonction :

	x	y
contenu initial	a	b
`x := !x + !y`	$a+b$	b
`y := !x - !y`	$a+b$	a
`x := !x - !y`	b	a

Sachant que le type *int* ne peut prendre qu'un nombre fini de valeurs dans un intervalle de la forme $[\![-n, n-1]\!]$ et que les additions et soustractions opèrent modulo $2n$ (avec $n = 2^{62}$ ou $n = 2^{30}$), on pourrait craindre que les opérations effectuées n'occasionnent une erreur, mais il n'en est rien, car l'exacte évolution du contenu des références est le suivant :

	x	y
contenu initial	a	b
`x := !x + !y`	$a+b \mod n$	b
`y := !x - !y`	$a+b \mod n$	$a \mod n$
`x := !x - !y`	$b \mod n$	$a \mod n$

Or si a et b appartiennent initialement à l'intervalle $[\![-n, n-1]\!]$ alors $a \mod n = a$ et $b \mod n = b$, donc la permutation est dans tous les cas correcte.

Exercice 2 La suite de Fibonacci est définie par les relations $f_0 = f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$ pour $n \geqslant 2$; nous allons poser $u_{-1} = 0$ et itérer la suite $u_n = (f_n, f_{n-1})$ qui vérifie les relations $u_0 = (1,0)$ et $u_n = \varphi(u_{n-1})$, où φ est la fonction définie par $\varphi(x,y) = (x+y, x)$:

```
let fib n =
  let ux = ref 1 and uy = ref 0 in
  for k = 1 to n do
    let a = !ux in ux := !ux + !uy ; uy := a done ;
  !ux ;;
```

Exercice 3 Nous allons utiliser une référence dont la valeur initiale est la liste vide, et dans laquelle nous allons un par un insérer en tête de liste les éléments de la liste à renverser :

```
let miroir l =
  let lst = ref l and tsl = ref [] in
    while not !lst = [] do tsl := (hd !lst)::!tsl ;
                           lst := tl !lst done ;
    !tsl ;;
```

C'est très laid, et on évitera par la suite de traiter de manière impérative les listes Caml.

Exercice 4 Dans un style fonctionnel, les deux fonctions demandées sont évidentes car elles se contentent de reprendre les formules mathématiques :

```
let rec f x = function
  | 0 -> 1
  | n -> x * (f x (n-1)) ;;
```

```
let rec g x = function
  | 0 -> 1
  | n when n mod 2 = 0 -> g (x*x) (n/2)
  | n -> x * g (x*x) (n/2) ;;
```

Dans un style impératif, la première de ces deux fonctions s'écrit :

```
let f x n =
  let p = ref 1 in
    for i = 1 to n do p := x * !p done ;
    !p ;;
```

À l'évidence, n multiplications sont effectuées.

Pour écrire la seconde fonction dans un style impératif, nous allons utiliser un accumulateur en considérant la fonction suivante :

$$\varphi(a,x,n) = \begin{cases} a & \text{si } n = 0 \\ \varphi(a,x^2,p) & \text{si } n = 2p \\ \varphi(x\times a,x^2,p) & \text{si } n = 2p+1 \end{cases}$$

et en observant que $x^n = \varphi(1,x,n)$. Ceci conduit à la définition suivante :

```
let g x n =
  let k = ref n and y = ref x and a = ref 1 in
  while !k > 0 do
    if !k mod 2 = 1 then a := !a * !y ;
    y := !y * !y ;
    k := !k / 2 done ;
  !a ;;
```

Pour compter le nombre $\alpha(n)$ de multiplications utilisées, il faut faire intervenir la décomposition en base 2 de l'entier n : $n = [b_{m-1}b_{m-2}\cdots b_0]_2$. Avec ces notations, $p = [b_{m-1}b_{m-2}\cdots b_1]_2$, et $\alpha(n) = \alpha(p) + 1 + b_0$. Ainsi, $\alpha(n) = m + \sum_{k=0}^{m-1} b_k$, ce qui conduit à l'encadrement : $m+1 \leqslant \alpha(n) \leqslant 2m$, soit : $\lfloor \log_2 n \rfloor + 2 \leqslant \alpha(n) \leqslant 2\lfloor \log_2 n \rfloor + 2$.

Exercice 5 Les références `u` et `v` doivent être à contenu entier, donc `f` est de type *int -> int -> int*.

On peut observer que la fonction `f` calcule le résultat de la fonction $f : (u,v) \longmapsto \begin{cases} u & \text{si } v = 0 \\ f(v, u \mod v) & \text{sinon} \end{cases}$ appliqué en (a,b). On reconnaît dans ces formules le principe de l'algorithme d'Euclide, donc $f(a,b)$ calcule le pgcd de a et de b.
Les formules ci-dessus conduisent immédiatement à la version fonctionnelle suivante (qu'on est en droit de préférer) :

```
let rec pgcd u = function
  | 0 -> u
  | v -> pgcd v (u mod v) ;;
```

Exercice 6 Pour définir la fonction `map_vect`, il faut commencer par créer un tableau de même taille que le tableau initial. Seule difficulté, les règles de typage strict de Caml nous imposent d'en fixer le type dès la création.

```
let map_vect f t =
  let n = vect_length t in
    if n = 0 then [||] else
      begin
        let r = make_vect n (f t.(0)) in
          for k = 1 to n-1 do r.(k) <- f t.(k) done ;
        r
      end ;;
```

La fonction **do_vect** est évidente :

```
let do_vect f t =
  for k = 0 to (vect_length t - 1) do f t.(k) done ;;
```

Exercice 7 La version impérative de la recherche de palindrome repose sur un parcours de la chaîne de caractères à l'aide d'une boucle conditionnelle :

```
let palindrome s =
  let n = string_length s in
  let rep = ref true and k = ref 0 in
    while !rep && !k < (n-1) / 2 do
      rep := s.[!k] = s.[n-1 - !k] ;
      k := !k + 1 done ;
    !rep ;;
```

Quant à la version fonctionnelle, elle utilise une fonction auxiliaire et le principe de l'évaluation paresseuse pour éviter de recréer de nouvelles chaînes de caractères :

```
let palindrome s =
  let rec aux = fun
    | i j when i >= j -> true
    | i j -> s.[i] = s.[j] && aux (i+1) (j-1)
  in aux 0 (string_length s - 1) ;;
```

Exercice 8

a) Nous avons $u_{p+1} = u_p + a_p b^p$, mais il serait déraisonnable d'utiliser une des fonctions de l'exercice 4 pour calculer b^p à chaque itération, car il suffit d'itérer la suite (u_p, b^p). On obtient donc ceci :

```
let applique p b =
  let n = vect_length p in
    if n = 0 then 0 else
      begin
        let u = ref p.(0) and x = ref 1 in
          for k = 1 to n-1 do
            x := !x * b ;
            u := !u + p.(k) * !x done ;
          !u
      end ;;
```

À l'évidence cette fonction effectue $(n-1)$ additions et $2(n-1)$ multiplications.

b) Il s'agit cette fois d'itérer la suite $v_p = \sum_{k=0}^{p} a_{n-1-k} b^{p-k}$, qui vérifie les relations $v_0 = a_{n-1}$ et $v_k = b v_{k-1} + a_{n-1-k}$ pour $k \in [\![1, n-1]\!]$. Nous avons alors $P(b) = v_{n-1}$.

```
let horner p b =
  let n = vect_length p in
    if n = 0 then 0 else
      begin
        let v = ref p.(n-1) in
          for k = 1 to n-1 do
            v := b * !v + p.(n-1-k) done ;
          !v
      end ;;
```

Cette fonction effectue toujours $(n-1)$ additions mais seulement $(n-1)$ multiplications.

c) Pour calculer $P(X+b)$, nous allons nous inspirer de la méthode de Hörner en itérant la suite de polynômes $V_p = \sum_{k=0}^{p} a_{n-1-k}(X+b)^{p-k}$ qui vérifie $V_0 = a_{n-1}$ et $V_k = (X+b)V_{k-1} + a_{n-1-k}$:

```
let translate p b =
  let n = vect_length p in
    if n = 0 then [||] else
      begin
        let q = make_vect n 0 in
          q.(0) <- p.(n-1) ;
          for k = 1 to n-1 do
            for j = k downto 1 do
              q.(j) <- q.(j) * b + q.(j-1) done ;
            q.(0) <- q.(0) * b + p.(n-1-k) done ;
          q
      end ;;
```

Exercice 9

a) Rappelons que si un nœud a pour numéro k, les numéros de ses deux fils sont $2k$ et $2k+1$. Ceci nous conduit à écrire une fonction vérifiant si chaque nœud ou feuille, à l'exception de la racine, a bien une étiquette supérieure à celle de son père :

```
let est_un_tas t =
  let n = vect_length t in
  let rep = ref true and k = ref (n-1) in
    while !rep && !k > 1 do rep := t.(!k) >= t.(!k/2) ; k := !k-1 done ;
    !rep ;;
```

Ou si on préfère une version fonctionnelle :

```
let est_un_tas t =
  let rec aux = function
    | 0 | 1 -> true
    | k     -> t.(k) >= t.(k/2) && (aux (k-1))
  in aux (vect_length t - 1) ;;
```

b) Le principe consiste à remonter l'élément placé au bout du tas, en le permutant avec son père, jusqu'à ce qu'il trouve sa place :

```
let swap_vect t i j =
  let m = t.(i) in t.(i) <- t.(j) ; t.(j) <- m ;;

let monter t =
  let n = vect_length t in
  let k = ref (n-1) in
    while t.(!k) < t.(!k/2) do swap_vect t !k (!k/2) ; k := !k/2 done ;;
```

c) Il s'agit maintenant de faire descendre un élément en le permutant avec le plus petit de ses fils, jusqu'à ce qu'il trouve sa place :

```
let rec descendre t k =
  let n = vect_length t in
  let f = if 2*k+1 >= n || t.(2*k) < t.(2*k+1) then 2*k else 2*k+1 in
  if f < n && t.(k) > t.(f) then begin swap_vect t k f ;
                                      descendre t f end ;;
```