Chapitre 5

Correction des exercices

Exercice 1 Nous allons bien entendu utiliser deux piles, l'une qui va recevoir les assiettes bleues, l'autre les rouge. Une fois la pile initiale vidée, elle sera remplie de nouveau en y déversant d'abord les assiettes bleues, puis les rouges.

```
let rec tri p b r = let x = pop p in
  match fst x with
    | Bleue -> push x b ; tri p b r
    | Rouge -> push x r ; tri p b r ;;

let rec empile a p = push (pop a) p ; empile a p ;;

let rangement p =
  let b = new () and r = new () in
  try tri p b r with
    | Empty -> try empile b p with
                 | Empty -> try empile r p with
                              | Empty -> () ;;
```

La fonction **`tri`** a pour objet de trier les assiettes de la pile **`p`** en deux piles : les assiettes bleues dans la pile **`b`**, les rouges dans la pile **`r`**. Une fois la pile **`p`** vide, l'exception **`Empty`** se déclenche.
La fonction **`empile`** transfère les assiettes de la pile **`a`** vers la pile **`p`** jusqu'au déclenchement de l'exception **`Empty`**.

Exercice 2 Le principe est simple : le sommet de la pile est mémorisé, les autres éléments sont déversés dans une pile auxiliaire, puis on reconstitue la pile sous la forme souhaitée.

```
let transvase p q =
  let rec aux () = push (pop q) p ; aux () in
    try aux () with Empty -> () ;;

let rotation p =
  let q = new () in
    try let a = pop p in
          transvase q p ;
          push a p ;
          transvase p q
    with Empty -> () ;;
```

La fonction **`transvase`** permet de transférer les éléments d'une pile vers une autre.
Pour effectuer la rotation inverse, on transvase la pile entière dans la pile auxiliaire, on mémorise alors le sommet de celle-ci, puis on reconstitue la pile sous la forme souhaitée.

```
let rotation_inv p =
  let q = new () in
    try transvase q p ;
        let a = pop q in
          transvase p q ;
          push a p
    with Empty -> () ;;
```

Exercice 3 Le type de la fonction **new** que nous allons écrire va différer de celui écrit en cours, car lorsqu'on utilise un tableau, il est nécessaire de connaître à l'avance le type d'objet qu'on va y ranger. Nous allons donc écrire une fonction de type *'a –> 'a file*, le paramètre d'entrée n'ayant comme seul rôle de préciser le type du contenu de la file :

```
let new x = {N = 100 ; Tab = make_vect 100 x ; Tete = 0 ; Queue = 0} ;;
```

La valeur de n choisie est bien entendu purement arbitraire.

La fonction **add** demande de vérifier que la file n'est pas pleine :

```
let add x f = match (f.Tete - f.Queue) mod f.N with
  | 1 -> raise Full
  | _ -> f.Tab.(f.Queue) <- x ; f.Queue <- (f.Queue + 1) mod f.N ;;
```

et la fonction **take**, que la file n'est pas vide :

```
let take f = match (f.Tete - f.Queue) with
  | 0 -> raise Empty
  | _ -> let k = f.Tete in f.Tete <- (k + 1) mod f.N ; f.Tab.(k) ;;
```

Exercice 4 Si on utilise une pile au lieu d'une file dans la parcours en largeur d'un arbre, on obtient le parcours en profondeur suivant l'ordre préfixe :

```
let parcours arbre =
  let pile = new () in
  push arbre pile ;
  let rec aux () = match (pop pile) with
    | Nil                   -> aux ()
    | Noeud (r, fils_g, fils_d) -> print_int r ;
                               push fils_d pile ;
                               push fils_g pile ;
                               aux ()
  in try aux () with Empty -> () ;;
```

On obtient le parcours infixe et le parcours suffixe en modifiant l'ordre de traitement.

– pour le parcours infixe :

```
push fils_d pile ;
print_int r ;
push fils_g pile ;
```

– pour le parcours suffixe :

```
push fils_d pile ;
push fils_g pile ;
print_int r ;
```

Exercice 5

a) Il suffit de suivre pas à pas le descriptif de l'énoncé :

```
let hamming n =
  let f2 = new () and f3= new () and f5 = new () in
  add 1 f2 ; add 1 f3 ; add 1 f5 ;
  for i = 1 to n do
    let x2 = peek f2 and x3 = peek f3 and x5 = peek f5 in
    let x = min x2 (min x3 x5) in
    print_int x ; print_char ` ` ;
    if x = x2 then ( let _ = take f2 in () ) ;
    if x = x3 then ( let _ = take f3 in () ) ;
    if x = x5 then ( let _ = take f5 in () ) ;
    add  (2*x) f2 ; add (3*x) f3 ; add (5*x) f5 ;
  done ;;
```

b) Un entier de Hamming k multiple à la fois de 3 et de 5 va apparaître dans les files **f3** et **f5**, mais il apparaîtra en premier dans cette dernière puisque $\frac{k}{5} < \frac{k}{3}$. Il ne faut donc ajouter un nombre de Hamming n dans la file **f3** que si n n'est pas multiple de 5.

Pour les mêmes raisons, le nombre de Hamming n ne sera rajouté à la file **f2** que s'il n'est ni multiple de 3, ni multiple de 5.

Ceci conduit à la version optimisée suivante :

```
let hamming n =
  let f2 = new () and f3= new () and f5 = new () in
  add 1 f2 ; add 1 f3 ; add 1 f5 ;
  for i = 1 to n do
    let x2 = peek f2 and x3 = peek f3 and x5 = peek f5 in
    let x = min x2 (min x3 x5) in
    print_int x ; print_char ' ' ;
    if x = x2 then ( let _ = take f2 in () ) ;
    if x = x3 then ( let _ = take f3 in () ) ;
    if x = x5 then ( let _ = take f5 in () ) ;
    add (5*x) f5 ;
    if x mod 5 <> 0 then ( add (3*x) f3 ;
                           if x mod 3 <> 0 then add (2*x) f2 ) ;
  done ;;
```

Exercice 6

a) La deuxième et la quatrième des permutations peuvent être engendrées, respectivement par les suites d'opérations : EEEDEDDD et EEEDEEEDEEDDEDDEDDEDD.

La première permutation ne peut être engendrée par une pile, car pour pouvoir dépiler 3 en premier, il faut avoir empilé d'abord 1, puis 2 ; mais il est alors impossible de dépiler 1 avant 2.

La troisième permutation ne peut non plus être engendrée par une pile, car pour pouvoir dépiler 7, il faut avoir empilé 6 ; mais 6 ne peut être empilé qu'après 1, et ne peut donc être dépilé après.

b) Supposons l'existence de $i < j < k$ tel que $a_j < a_k < a_i$.

Puisque $i < j < k$, a_i doit être dépilé en premier, puis a_j et enfin a_k. Mais la seconde inégalité implique que lorsque a_i est dépilé, a_j et a_k se trouvent déjà dans la pile, a_j étant le plus bas. Il ne peut donc être dépilé avant a_k, et la permutation n'est donc pas engendrable.

c) Nous allons utiliser un accumulateur qui va mémoriser la valeur i du plus grand entier a avoir été empilé. Lorsqu'il va falloir dépiler l'entier j, nous allons commencer par empiler les entiers compris entre $i+1$ et j (si $i < j$) puis dépiler j s'il se trouve au sommet de la pile. Dans le cas contraire, c'est que la permutation n'est pas engendrable.

```
let engendrable =
  let p = new () in
  let rec aux i = function
    | []   -> true
    | j::q -> for k = i+1 to j do push k p ; print_char 'E' ; done ;
              match (pop p) with
                | k when k=j -> print_char 'D' ; aux (max i j) q
                | _ -> false
  in aux 0 ;;
```

d) La fonction précédente ne permet pas d'engendrer une permutation lorsqu'au moment de dépiler j, ce dernier ne se trouve pas au sommet de la pile. Dans ce cas, il suffit de stocker temporairement dans une seconde pile les éléments situés au dessus de lui, puis de les faire réintégrer la pile initiale une fois j dépilé.

```
let transfert q p =
  let rec aux () = push (pop q) p ; print_char `d` ; aux () in
  try aux () with Empty -> () ;;

let rec cherche j p q = match pop p with
    | k when k = j -> print_char `D` ; transfert q p
    | k            -> push k q ; print_char `e` ; cherche j p q ;;

let génération =
  let p = new () and q = new () in
  let rec aux i = function
    | []   -> ()
    | j::r -> for k = i+1 to j do push k p ; print_char `E` ; done ;
              cherche j p q ;
              aux (max i j) r
  in aux 0 ;;
```

La fonction **cherche** empile dans **q** les éléments de **p** qui se trouvent au dessus de **j** ; une fois trouvé, les éléments de **q** sont de nouveau renvoyés dans **p**. ces opérations d'empilement et de dépilement dans la pile **q** sont codées par les lettres *e* et *d*. Voici par exemple ce que donne la génération des deux permutations de la question a. qui n'étaient pas engendrables à l'aide d'une seule pile :

```
# génération [3; 1; 2] ;;
EEEDeDdD- : unit = ()
# génération [4; 5; 3; 7; 2; 1; 6] ;;
EEEEDEDDEEDeDdeDdD- : unit = ()
```