Correction : feuille de TD n°5

MPSI Lycée Clemenceau : option informatique

Mai 2025

Remarques : quelques fonctions utilises pour l'étude des piles.

- lecture d'une pile d'entiers qui ne modifie pas la pile :

```
let rec lecture p = match Stack.is_empty p with
                | true -> ()
                | false -> let x = Stack.pop p in print_int x;
             lecture p; Stack.push x p;;
```

- on met une pile dans une liste afin de la voir, sans modifier la pile. Cette version a l'intérêt d'être polymorphe :

```
let lecture2 p = let rec aux p lst = match Stack.is_empty p with
             | true -> lst
             | false -> let x = Stack.pop p in
                        let lt = x::lst in
                        let ll = aux p lt in Stack.push x p;ll;
          in  aux p [];;
```

Exercice 1 : Ecrire une fonction `rotation` qui place le premier élément de la pile en dernière position.

Correction : La fonction transvase permet de mettre une pile dans une autre (en inversant l'ordre bien évidemment).

```
let rotation p =
     if Stack.is_empty p then failwith "pile vide"
     else begin
      let a = Stack.pop p in
       let rec transvase p1 p2 =
         if Stack.is_empty p1 then ()
         else begin let b = Stack.pop p1 in
             Stack.push b p2;
             transvase p1 p2
             end
       in
        let q = Stack.create ()
        in
          transvase p q;
          Stack.push a p;
          transvase q p
         end;;
```

Autre méthode :

```
let rotation p =
    let rec aux p base = match Stack.is_empty p with
        | true -> Stack.push base p
        | false -> let x = Stack.pop p in aux p base ; Stack.push x p
    in match Stack.is_empty p with
        | true -> ()
        | false -> let base = Stack.pop p in aux p base ;;
```

Remarque : la fonction aux fait la même chose que transevase (dans la première fonction) deux fois mais ce n'est pas visible : c'est contenu dans les appels successifs avant d'empiler. J'insiste que cela utilise une pile supplémentaire non visible dans la fonction.

Exercice 2 :

1) Ecrire une fonction `retourne` qui renverse une pile.

Correction : on utilise deux piles intermédiaires et la fonction transvase du premier exercice :

```
let retourne p =
   let  rec transvase p1 p2 =
          if Stack.is_empty p1 then ()
          else begin let b = Stack.pop p1 in
                Stack.push b p2;
                transvase p1 p2 end
   in let q = Stack.create () and r = Stack.create() in
      transvase p q; transvase q r; transvase r p;;
```

2) Ecrire une fonction `split_parite` qui sépare une pile en deux piles (l'une contenant les éléments d'indice pair, l'autre ceux d'indice impair).

Correction : on remplie deux piles successivement. On s'arrête lorsqu'il ne reste plus aucun élément dans la pile initiale ou une seul que l'on met dans la première pile, sinon on continue les appels récursifs.

```
let split_parite p =
    let pp = retourne p in
      let rec aux p p1 p2 =
         if estvide p then (p1,p2)
         else let a = pop p in
               if estvide p then (push a p1; (p1,p2))
               else (push a p1; push (pop p) p2; aux p p1 p2);
      in aux pp (newpile ()) (newpile ());;
```

Remarque : à noter l'utilisation des parenthèses à la suite du `then` ou du `else` afin d'éviter l'utilisation de `begin ... end`.

Exercice 3 : *Avec l'utilisation de la librairie* ***`Stack`***

On dispose d'une pile d'assiettes bleues ou rouges numérotées disposées dans le désordre. Comment procéder pour former une pile dans laquelle les assiettes bleues sont situées sous les assiettes rouges, mais en faisant en sorte que pour chacune des deux couleurs l'ordre relatif ne soit pas modifié ? (Autrement dit, si l'assiette bleue i est située sous l'assiette bleue j dans la pile initiale, ce sera toujours le cas dans la pile finale.)

On définit les types :

```
# type couleur = Bleue | Rouge and assiette = couleur * int ;;
```

Rédiger une fonction de type `assiette t -> unit` qui range une pile d'assiette en disposant les assiettes bleues sous les assiettes rouges.

Correction : remarque pour commencer : le type assiette étant en fait un couple Ocaml n'affichera pas systématiquement que c'est une assiette mais cela n'a pas d'importance pour le programme.

```
type couleur = Bleue | Rouge and assiette = couleur * int ;;

(*avec gestion de l'exception*)

let rangement p =
    let rec tri p b r = let x = Stack.pop p in
          match fst x with
             | Bleue -> Stack.push x b; tri p b r
             | Rouge -> Stack.push x r; tri p b r in
    let rec empile a p = Stack.push (Stack.pop a) p ; empile a p in
    let b = Stack.create () and r = Stack.create () in
    try tri p b r with
      | empty -> try empile b p with
         | empty -> try empile r p with
            | empty -> () ;;

(*sans gestion de l'exception*)
let rangement2 p =
    let rec tri2 p b r = if not(p=Stack.create()) then let x = Stack.pop p in
        match fst x with
           | Bleue -> Stack.push x b; tri2 p b r
           | Rouge -> Stack.push x r; tri2 p b r in
    let rec empile2 a p = if not(Stack.is_empty a) then
             begin Stack.push (Stack.pop a) p ;
                   empile2 a p
             end in
    let b = Stack.create() and r = Stack.create () in
      if not(Stack.is_empty p) then tri2 p b r ;
      if not(Stack.is_empty b) then empile2 b p ;
      if not(Stack.is_empty r) then empile2 r p ;;


(*remplissage aléatoire d'une pile d'assiettes*)
let remp n = let t = Stack.create() in
         let r = ref 1 and b = ref 1 in
         for i = 1 to n do
            match Random.int(2) with
              | 0 -> Stack.push (Rouge,!r) t; incr(r);
              | _ -> Stack.push (Bleue,!b) t; incr(b);
         done;
         t;;
```

Exercice 4 : A l'aide des fonctions primitives du type pile, écrire une fonction `eval` qui réalise, de façon impérative, l'évaluation d'une expression arithmétique postfixée. Celle-ci est codée sous forme d'un tableau dont les éléments sont du type chaine de caractères.

Correction : cf cours

```
let eval tab = let p = Stack.create () in
               let n = Array.length tab in
               for i = 0 to n-1 do
                  match tab.(i) with
                    | "+" -> let a = Stack.pop p in
                             let b = Stack.pop p in
                             Stack.push (b + a) p
                    | "-" -> let a = Stack.pop p in
                             let b = Stack.pop p in
                             Stack.push (b - a) p
                    | "*" -> let a = Stack.pop p in
                             let b = Stack.pop p in
                             Stack.push (b * a) p
                    | "/" -> let a = Stack.pop p in
                             let b = Stack.pop p in
                             Stack.push (b / a) p
                    | _ -> Stack.push (int_of_string (tab.(i))) p
                   done;
                Stack.pop p;;
```

Exercice 5 : On dit qu'une permutation $(a_1\, a_2\, \ldots\, a_n)$ de $(1\, 2\, \ldots\, n)$ peut être engendrée par une pile lorsque il est possible, à partir de la séquence d'entrée $(1\, 2\, \ldots\, n)$ et d'une pile (initialement vide), de produire la séquence de sortie $(a_1\, a_2\, \ldots\, a_n)$ en utilisant les opérations suivantes :

— empiler l'élément suivant de la séquence d'entrée ;

— ou dépiler le sommet de la pile et l'imprimer à l'écran.

Par exemple, si E et D désignent respectivement chacune des deux opérations permises, la permutation $(2\, 3\, 1)$ est engendrée par la suite d'opérations : EEDEDD.

1) Parmi les permutations suivantes, lesquelles peuvent être engendrées par une pile ?

$$(3\ 1\ 2) \quad (3\ 4\ 2\ 1) \quad (4\ 5\ 3\ 7\ 2\ 1\ 6) \quad (3\ 5\ 7\ 6\ 8\ 4\ 9\ 2\ 10\ 1)$$

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

2) Montrer que s'il existe un triplet $(i, j, k) \in [\![1, n]\!]^3$ tel que $i < j < k$ et $a_j < a_k < a_i$, alors la permutation $(a_1\, a_2\, \ldots\, a_n)$ n'est pas engendrable par une pile.

Correction : 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.

3) Écrire une fonction `Caml` déterminant si une permutation peut être engendrée par une pile. Dans le cas d'une réponse positive, la fonction affichera la suite d'opérations permettant de la produire. Les permutations seront représentées par le type `int list`.

Correction : 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 lst = let pp = Stack.create() in
        let rec aux i lst = match lst with
            | [] -> true
            | j::q -> for k = i +1 to j do Stack.push k pp ; print_char 'E' done ;
                          match (Stack.pop pp) with
                             | k when k=j -> print_char 'D' ; aux  (max i j) q
                             | _ -> false
               in aux 0 lst;;
```

4) Montrer enfin que toute permutation peut être engendrée à l'aide de deux piles, et rédiger la fonction `Caml` correspondante.

Correction : 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 () = Stack.push (Stack.pop q) p; print_char 'd'; aux () in
    try aux () with empty -> ();;

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

let generation lst = let p = Stack.create () and q = Stack.create () in
     let rec aux i lst = match lst with
          | [] -> ()
          | j::r -> for k = i+ 1 to j do Stack.push k p ; print_char 'E' ; done ;
                    cherche j p q;
                    aux (max i j) r
     in aux 0 lst;;
```

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.

Exercice 6 : Les nombres de Hamming sont les entiers naturels non nuls dont les seuls facteurs premiers éventuels sont 2, 3 et 5 :

1; 2; 3; 4 5; 6; 8; 9; 10; 12; 15; 16; 18; 20; 24; 25; 27; 30; ...

Le but de cet exercice est de les générer de manière croissante. Évidemment, on peut parcourir un à un tous les entiers en testant à chaque fois si ceux-ci sont des entiers de Hamming, mais

cette démarche montre vite des limites (songez que le 2000e entier de Hamming est égal à 8 100 000 000 et le 2001e à 8 153 726 976 : il faudrait tester plus de 53 millions de nombres avant d'augmenter notre liste d'un élément !).

On adopte donc la démarche suivante : on utilise trois files f_2, f_3, f_5 contenant initialement le nombre 1, et on suit la démarche suivante :

1. on détermine le plus petit des trois têtes de file, que l'on note k et que l'on imprime à l'écran ;
2. on retire cet élément des files où il se trouve ;
3. on insère en queue des files f2, f3 et f5 les entiers 2k, 3k et 5k.

Vous l'avez compris : cette démarche utilise le fait que tout nombre de Hamming différent de 1 est le produit par 2, 3 ou 5 d'un nombre de Hamming plus petit.

1) Rédiger une fonction Caml permettant l'affichage des n premiers nombres de Hamming.
Correction : Il suffit de suivre pas à pas le descriptif de l'énoncé :
rappel : `Queue.peek` donne la valeur de la tête de la file sans la retirer de la file.

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

2) L'inconvénient de la démarche précédente est que le même nombre peut se retrouver dans plusieurs des trois files. Modifier votre fonction pour que cela ne soit plus le cas.
Correction : 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 = Queue.create () and f3= Queue.create () and f5 = Queue.create () in
    Queue.add 1 f2 ; Queue.add 1 f3 ; Queue.add 1 f5 ;
    for i = 1 to n do
        let x2 = Queue.peek f2 and x3 = Queue.peek f3 and x5 = Queue.peek f5 in
        let x = min x2 (min x3 x5) in
        print_int x ; print_char ' ' ;
        if x = x2 then (let _ = Queue.take f2 in () ) ;
        if x = x3 then (let _ = Queue.take f3 in () ) ;
        if x = x5 then (let _ = Queue.take f5 in () ) ;
        Queue.add (5*x) f5 ;
        if x mod 5 <> 0 then begin
             Queue.add (3*x) f3 ;
             if x mod 3 <> 0 then Queue.add (2*x) f2 ; end;
    done ;;
```