Files, Piles, Boucles, et Tableaux en OCaml

Florian Bourse

On rappelle la syntaxe des boucles en OCaml par 2 exemples concis :

boucle for

```
for i = 0 to 5 do
  print_int i; print_newline ()
done
```

boucle while

```
while true do
  let l = read_line () in print_string l
done
```

1 Files

Une file est une structure de donnée abstraite qui possède 3 fonctions principales : `Créer`, `Enfiler` (*enqueue*), et `Défiler` (*dequeue*). La particularité d'une file est son fonctionnement FIFO (First In First Out), c'est-à-dire que l'élément retiré de la file par `Défiler` est le premier élément a avoir été inséré par `Enfiler` qui n'a pas encore été retiré.
En OCaml, une implémentation de cette structure est donnée par le module `Queue`, dont on rappelle l'interface de programmation (API) ici :

- `Queue.create : unit -> 'a Queue.t`
  Return a new queue, initially empty.
- `Queue.push : 'a -> 'a Queue.t -> unit`
  `Queue.push x q` adds the element `x` at the end of the queue `q`.
- `Queue.pop : 'a Queue.t -> 'a`
  `Queue.pop q` removes and returns the first element in queue `q`, or raises `Queue.Empty` if the queue is empty.

Le module `Queue` propose aussi la fonction suivante :

- `peek : 'a Queue.t -> 'a`
  `peek q` returns the first element in queue `q`, without removing it from the queue, or raises `Queue.Empty` if the queue is empty.

Question 1. Écrire la fonction `random_queue : int -> int -> int Queue.t` telle que `random_queue n k` renvoie une nouvelle file, possédant `n` entiers choisis aléatoirement entre `0` inclus et `k` exclus. On rappelle qu'un tel nombre peut être tiré avec `Random.int k`, en ayant au préalable initialisé le générateur aléatoire avec `Random.self_init ()` (une seule initialisation suffit pour tout le programme).

Question 2. Écrire une fonction `is_empty : 'a Queue.t -> bool` qui permet de tester si une file est vide.

Question 3. Écrire une fonction `transfer : 'a Queue.t -> 'a Queue.t -> unit` telle que `transfer q1 q2` ajoute tous les éléments de la file `q1` à la fin de la file `q2`.

Question 4. Écrire une fonction `print_queue : int Queue.t -> unit` qui permet d'afficher les éléments d'une file d'entiers, un entier par ligne.

Question 5. Écrire une fonction `queue_length : 'a Queue.t -> int` qui permet de déterminer la taille d'une file. La file n'est pas changée.

Question 6. Écrire une fonction `iter : ('a -> unit) -> 'a Queue.t -> unit`, telle que `iter f q` applique la fonction `f` à tous les éléments de la file `q`. La file n'est pas changée.

Question 7. Réécrire la fonction `print_queue` en utilisant la fonction `iter`.
On pourra utiliser la construction `fun x -> ...` pour définir une fonction sans lui donner d'identifiant.

Question 8. Écrire la fonction `fold : ('b -> 'a -> 'b) -> 'b -> 'a Queue.t -> 'b` telle que `fold f acc q` renvoie `f (... (f (f acc a1) a2) ... ) an` où les `ai` sont les éléments de la file `q` dans l'ordre de sortie. La file n'est pas changée.

Question 9. Réécrire la fonction `queue_length` en utilisant la fonction `fold`.

Question 10. Écrire une fonction `sum : int Queue.t -> int` qui calcule la somme des éléments d'une file d'entiers. La file n'est pas changée.

Question 11. Écrire une fonction `take_min : int Queue.t -> int` telle que `take_min q` renvoie le plus petit élément de la file `q`. Cet élément est retiré de la file. L'ordre des autres éléments n'est pas modifié. On pourra décomposer cette fonction en deux étapes.

2 Piles

Une pile est une structure de donnée abstraite qui possède 3 fonctions principales : `Créer`, `Enpiler` (*push*), et `Dépiler` (*pop*). La particularité d'une pile est son fonctionnement LIFO (Last In First Out), c'est-à-dire que l'élément retiré de la pile par `Dépiler` est le dernier élément a avoir été inséré par `Enpiler` qui n'a pas encore été retiré.
En OCaml, une implémentation de cette structure est donnée par le module `Stack`, dont on rappelle l'interface de programmation (API) ici :

- `Stack.create : unit -> 'a Stack.t`
  Return a new stack, initially empty.
- `Stack.push : 'a -> 'a Stack.t -> unit`
  `Stack.push x s` adds the element `x` at the end of the stack `s`.
- `Stack.pop : 'a Stack.t -> 'a`
  `Stack.pop s` removes and returns the first element in stack `s`, or raises `Stack.Empty` if the stack is empty.

Question 12. Écrire la fonction `peek : 'a Stack.t -> 'a` telle que `peek s` renvoie la premier élément de la pile `s`, sans la modifier ou lève l'exception `Stack.Empty` si la pile est vide.

Question 13. Écrire une fonction `contains : 'a -> 'a Stack.t -> bool` qui permet de tester si un élément `x` apparaît dans une pile. La pile n'est pas modifiée.

Question 14. Écrire une fonction `copy : 'a Stack.t -> 'a Stack.t` qui permet de faire une copie d'une pile.

Un problème algorithmique classique consiste à trier une pile de pancakes, la seule opération permettant de modifier la pile étant de retourner le haut de la pile (imaginez insérer une spatule à un endroit de la pile et retourner son contenu sur le haut de la pile). Essayons de modéliser cette opération par une pile.

Question 15. Pour retourner les i premiers éléments d'une pile, nous allons les retirer de la pile, avant de les remettre au dessus. Est-il plus judicieux de les stocker dans une pile ou dans une file en attendant de les remettre ?

Question 16. Écrire une fonction `flip : int -> 'a Stack.t -> unit` telle que `flip i s` retourne les `i` premiers éléments de la pile `s`.

Question 17. Écrire une fonction `full_flip : 'a Stack.t -> unit` qui permet de retourner complètement une pile.

Question 18. Écrire une fonction `is_sorted : 'a Stack.t -> bool` qui permet de tester si une pile est triée. La pile n'est pas changée.

Question 19. Écrire une fonction `argmax : 'a Stack.t -> int` qui donne l'indice du plus grand élément de la pile.

Question 20. Écrire une fonction `sort : 'a Stack.t -> unit` qui permet de trier une pile en mettant les éléments les plus grand en dessous et les éléments les plus petit au dessus.

3 Tableaux, Matrices

En OCaml, les tableaux sont implémentés par le type `'a array`. Deux fonctions premettent de créer des tableaux :

— `Array.make : int -> 'a -> 'a array`
`Array.make n x` renvoie un nouveau tableau de taille `n`, indicé par les entiers de `0` à `n-1` dont toutes les valeurs sont égales à `x`.

— `Array.init : int -> (int -> 'a) 'a array`
`Array.init n f` renvoie un nouveau tableau de taille `n`, indicé par les entiers de `0` à `n-1` dont la case `i` contient `f i`.

La fonction `Array.make` possède aussi une variante

```
Array.make_matrix : int -> int -> 'a -> 'a array array
```

qui permet de créer un tableau à 2 dimensions. `Array.make_matrix dimx dimy e` créer un nouveau tableau de taille `dimx` dont chaque élément est un tableau de taille `dimy` dont tous les éléments sont `e`.

On accède et on modifie les éléments d'un tableau avec les syntaxes :

— `a.(n)` pour accéder à l'élément du tableau `a` indicé par `n`. C'est un alias pour `Array.get a n`

— `a.(n) <- x` pour affecter la valeur `x` à l'élément du tableau `a` indicé par `n`. C'est un alias pour `Array.set a n x`.

On peut connaître la taille d'un tableau grâce à la fonction :

— `Array.length : 'a array -> int`.

Question 21. Dans une fête, on cherche à savoir si il y a une superstar. Une superstar est une personne que tout le monde connaît mais qui ne connaît personne. Le problème est décrit par une matrice carrée `a : bool array array` telle que `a.(i).(j)` vaut `true` si la personne `i` connaît la personne `j`, et `false` sinon. On ne prendra pas en compte les éléments de la diagonale.
Déterminer la complexité de votre solution.

On pourra tester avec les matrices suivantes :

ex1

```
[|
  [|0;0;1;0|];
  [|0;0;1;0|];
  [|0;0;0;0|];
  [|0;0;1;0|];
|]
```

ex2

```
[|
  [|0;0;1;0|];
  [|0;0;1;0|];
  [|0;1;0;0|];
  [|0;0;1;0|];
|]
```