TP 7 : Programmation impérative en OCaml

Le but de ce TP est de vous proposer un entraînement à la programmation impérative en OCaml. Il est à terminer pour la prochaine séance (j'essaye d'en caler une la semaine prochaine) : vous devez vous y plonger et réussir à faire la plupart des questions des Exercices 1 à 5. Après vos questions éventuelles au début de la prochaine séance, je considère que les principes et la syntaxe de la programmation impérative en OCaml est connue, et on va intensivement l'utiliser par la suite.
Dans ce TP, on va manipuler les `ref`, `unit`, `mutable`, `array`. Si l'un de ces mots ne vous dit rien → **??** : il faut faire ce TP en ayant ce cours sous les yeux (ou à côté). Vous disposez aussi d'une fiche de syntaxe rapide comprenant la syntaxe d'OCaml impérative et fonctionnelle sur le cahier de prépas.
Après ça, à part quelques opérateurs que l'on n'utilisera pas (par exemple `|>` ou encore `@@`), vous pourrez lire n'importe quel bout de code OCaml et interpréter ce qu'il fait. Un bon exercice peut être de regarder l'implémentation d'un module ou d'un autre en OCaml : par exemple, allez voir l'implémentation du module `Queue` et faire un dessin pour deviner comment elle est implémentée (on verra au prochain chapitre l'implémentation des files).
Vous avez un avantage absolu par rapport aux MP2I d'il y a quelques mois : vous avez déjà codé en impératif pendant plusieurs mois, en Python. Si vous êtes bloqués non pas sur la syntaxe, mais sur la stratégie pour résoudre une question, vous aurez sans doute moins de mal à l'écrire d'abord en Python puis à traduire en OCaml.
Un conseil important enfin : veillez à vérifier systématiquement que le type de vos valeurs / fonctions correspondent bien à l'énoncé avant de passer à la suite.

Exercice 1 – Manipulation d'un enregistrement modifiable

On va reprendre un exercice déjà fait en C : manipuler un enregistrement représentant un nombre complexe. On utilisera le type suivant.

```OCaml
type complexe = {
  mutable re : float; mutable im : float
}
```

▶ **Question 1** Écrire les fonctions `partie_reelle : complexe -> float` et `partie_imaginaire : complexe -> float` renvoyant la partie réelle et imaginaire d'un complexe. *Indication : ici, le fait qu'on aie rajouté le mot-clé* `mutable` *ne change rien à la syntaxe.*

▶ **Question 2** Écrire une fonction `conjugue : complexe -> unit` qui modifie en place un complexe pour qu'il devienne son conjugué.

On veillera bien à tester nos fonctions, notamment en vérifiant qu'elle a bien exactement le type attendu.

▶ **Question 3** Écrire une fonction `reinitialiser : complexe -> unit` qui change en place le complexe en entrée pour qu'il soit égal à $0 \in \mathbb{C}$.

▶ **Question 4** Écrire une fonction `norme : complexe -> float` qui à un complexe $z \in \mathbb{C}$ renvoie $|z|$.

▶ **Question 5** Écrire une fonction `incremente : complexe -> unit` qui modifie le complexe en entrée en y additionnant 1.

▶ **Question 6** Écrire la fonction `multiplier_par : complexe -> complexe -> unit` modifiant la première entrée pour qu'elle devienne égale au produit des deux entrées : après avoir appliqué `multiplier_par z1 z2`, `z1` est modifié pour contenir la valeur $z_1 \times z_2$.

Exercice 2 – Enchaîner des instructions en OCaml

En OCaml, on parle souvent d'*instructions* pour décrire les expressions de type `unit`. C'est le cas par exemple de l'expression `print_int 10`.

▶ **Question 1** Écrire une fonction `f : int -> unit` qui affiche un entier et reviens à la ligne. *Indication : un test de cette fonction dans* `ocaml/utop` *doit donner :*

```OCaml
# f 10;;
10
- : unit = ()
(* et pas : *)
10- : unit = ()
```

▶ **Question 2** Écrire une fonction `g : int -> float -> unit` qui prend un entier, un flottant et qui imprime le couple des arguments :

```OCaml
# g 10 3.5;;
(10, 3.5)
- unit = ()
```

On pourra utiliser toutes les fonctions `print_...` présentées dans le **??**.

▶ **Question 3** Déduire de la fonction `f` une fonction `affiche_liste_entiers : int list -> unit` qui affiche le contenu d'une liste avec un élément par ligne. *On pourra utiliser la fonction* `List.iter : (int -> unit) -> int list -> unit` *telle que l'appel* `List.iter g l` *exécute les instructions* `g l0; g l1; ...`.

Exercice 3 – Manipuler des références

▶ **Question 1** Écrire une fonction `affiche_ref_entier : int ref -> unit` qui affiche l'entier référencé par son argument.

▶ **Question 2** Écrire une fonction `echange_references : 'a ref -> 'a ref -> unit` qui échange la valeur contenue dans les deux références en entrée.

On rappelle que l'on crée une référence par la fonction `ref : 'a -> 'a ref`. ▶ **Question 3** Écrire une fonction `creer_ref : int -> int ref` qui crée une référence contenant l'argument, incrémente cette référence de 1 et renvoie cette référence.

▶ **Question 4** Tester cette fonction sur une variable `let n = 10`. Que remarque-t-on ? En quoi ce comportement est différent de créer une référence d'un entier en C avec la syntaxe `int* p = &n;` ?

▶ **Question 5** Créer une fonction `incrementer : int ref -> unit` qui incrémente la valeur contenue dans une référence entière.

▶ **Question 6** Même question pour `incrementer_flottant : float ref -> unit`.

▶ **Question 7** Écrire une fonction `modifie_premier_element : ('a * 'b) ref -> 'a -> unit` qui prend une référence d'un couple en entier et modifie le premier élément de ce couple. Même question pour `modifie_second_element : ('a * 'b) ref -> 'b -> unit`.

Exercice 4 – Écrivons des boucles

▶ **Question 1** Écrire une fonction `somme_entiers : int -> int` qui calcule la somme des entiers entre 0 et n, en utilisant une boucle `for` et en modifiant une référence à chaque tour de boucle.

▶ **Question 2** Modifier votre fonction pour qu'elle affiche à chaque tour la valeur calculée.

On rappelle l'algorithme d'Euclide calculant le PGCD de deux entiers :

OCaml
```
let rec euclide a b =
  if b = 0 then a
  else euclide b (a mod b)
```

On remarque que l'on peut voir cette fonction comme une boucle `while` : avec si `a` et `b` deux variables modifiables, tant que `b` est différent de zéro, on modifie `a` en `b` et `b` en `a mod b`.

▶ **Question 3** Implémenter cette stratégie à l'aide d'une boucle `while` : on crée deux références contenant initialement `a` et `b`, puis tant que la seconde référence est non nulle, on modifie les deux références comme décrit ci-dessus. *On fera attention à tester cette fonction.*

▶ **Question 4** Écrire une fonction `retourner_imperatif : 'a list -> 'a list` qui renvoie la liste en entrée retournée avec la stratégie suivante :

- on crée deux références de type `'a list ref`, la première contenant initialement l'entrée et la seconde contenant initialement la liste vide.
- À l'aide d'une boucle `while`, tant que la liste référencée par la première référence est non vide, on prend sa tête `h` et on modifie la seconde référence pour y ajouter `h` en tête.
- Quand la première référence contient une liste vide, on s'arrête et on renvoie le contenu de la seconde référence.

▶ **Question 5** Comparer votre code à celui de la fonction `retourner` récursive écrite dans le TP 2 ou 3. Qu'en pensez-vous ?

Exercice 5 – Manipulation de tableaux

Pour compléter votre panorama des structures impératives de base en OCaml, on va aussi travailler la manipulation des tableaux en OCaml.

▶ **Question 1** Écrire une fonction `element_zero : 'a array -> 'a` renvoyant l'élément d'indice zéro du tableau.

▶ **Question 2** Écrire une fonction `nombre_occurences : 'a array -> 'a -> int` tel que `nb_occurences t x` renvoie le nombre de fois que l'on trouve `x` dans le tableau `t`.

▶ **Question 3** Écrire une fonction `somme_bornes : float array -> int -> int -> float` tel que l'appel `somme_bornes t i j` calcule la somme des éléments du tableau compris entre l'indice `i` inclus et `j` exclus.

Le type des éléments que l'on met dans un tableau est `'a` : ainsi, ça peut être n'importe quoi : ça peut être des entiers/flottants/booléens, mais aussi d'un type somme, d'un type enregistrement, voire un tableau de fonctions...Et OCaml le gère tranquillou bilou.

▶ **Question 4** Écrire une fonction `indice_min : 'a array -> int` qui prend un tableau en entrée et renvoie l'indice de son minimum. On lèvera une exception si le tableau est vide (ce qui est possible !).

▶ **Question 5** Écrire une fonction `echange : 'a array -> int -> int -> unit` tel que `echange t i j` échange les valeurs `t.(i)` et `t.(j)`.

▶ **Question 6** En déduire une fonction `tri_selection : 'a array -> unit` qui implémente le tri par sélection en place.

Quelques fonctions pêle-mêle :

▶ **Question 7** Écrire une fonction `sommes_cumulees : float array -> float array` tel que si l'entrée est un tableau de flottants $[T_0, \dots, T_{n-1}]$, on renvoie le tableau contenant à l'indice i les sommes partielles des éléments de T entre 0 et i inclus.

▶ **Question 8** Écrire une fonction `moyenne : float array -> float` qui calcule la moyenne d'un tableau de flottants.

▶ **Question 9** Écrire une fonction `array_to_list : 'a array -> 'a list` qui prend un tableau en entrée et renvoie une liste comprenant tous les éléments du tableau dans le même ordre. Par exemple, `array_to_list [|1; 5; 2; 10|] = [1; 5; 2; 10]`.

▶ **Question 10** Écrire une fonction `write_list_in_array : 'a list -> 'a array -> int -> unit` tel que l'appel `write_list_in_array l t k` écrit les éléments de `l` dans l'ordre dans `t` en commençant à l'indice `k`. On lèvera une exception si la liste est trop longue. Par exemple, si `t = [|1; 5; 2; 10|]`, l'appel `write_list_in_array [3; -2] t 1` modifie le tableau en `t = [|1; 3; -2; 10|]`.

▶ **Question 11** En déduire la fonction réciproque : `list_to_array : 'a list -> 'a array`. *Dans cette fonction, il est parfaitement acceptable de parcourir deux fois la liste.*

▶ **Question 12** Écrire une fonction `map : ('a -> 'b) -> 'a array -> 'b array` qui prend en entrée et une fonction et qui renvoie le tableau des images des éléments du tableau par cette fonction. Par exemple :

```
# map (fun x -> x *. 2.) [|3.2; 5.4; 0.|];;
- : float array = [|6.4; 10.8; 0.|]
```
OCaml

Je pense que toutes les fonctions précédentes sont accessibles à toute la classe. Je vous demande de vous concentrer sur leur programmation, et de ne pas hésiter à collaborer puis à me poser des questions pour y répondre. Si vous avez plusieurs solutions, mais que vous avez du mal à les comparer, n'hésitez pas à me demander un éclairage.

Les questions suivantes demandent de réfléchir un peu plus : c'est juste pour le plaisir de coder des algos astucieux ! J'en proposerais une correction peu après la rentrée.

Exercice 6

▶ **Question 1** Quelle est la complexité de votre fonction `sommes_cumulees` ? En écrire éventuellement une version en $\mathcal{O}(n)$, avec n la taille du tableau en entier.

▶ **Question 2** (Extrait des épreuves régionales de Prologin 2014) : *Félicitations, vous êtes arrivés au dernier palier de l'épreuve machine d'aujourd'hui ! Il ne vous reste plus que quelques minutes avant la fin, et vous venez de coder l'algorithme final : un shuffle.*
Après avoir lu un énoncé absurde pendant plusieurs minutes, vous avez en effet réussi à comprendre qu'on vous demandait de programmer un algorithme qui mélange aléatoirement une liste. Cependant, vous n'avez plus de temps pour le tester avant la fin de l'épreuve, c'est pourquoi vous décidez de revenir dans le temps pour modifier le deuxième exercice de votre ancien "vous", afin qu'il code un algorithme vérifiant que la liste qu'on lui donne en entrée est bien un réarrangement de la liste d'origine.
Écrire une fonction `est_rearrangement : 'a list -> 'a list -> bool` répondant à la question, de complexité temporelle au plus en $\mathcal{O}(n \ln n)$ (même s'il est possible de faire moins).

▶ **Question 3** Écrire une fonction `doublon_opt : int array -> int option` vérifiant si un tableau de taille n d'entiers entre 0 et $n-1$ contient un doublon : si oui, on en renvoie un avec `Some n`, sinon on renvoie `None`. On vise une complexité temporelle linéaire et spatiale constante, et on ne suppose pas que le tableau reste inchangé.

On se donne une matrice à n lignes et p, représentée par un tableau de tableaux de type `'a array array`. La syntaxe `m.(i).(j)` permet d'accéder à la case d'indice `(i,j)` de cette matrice. On suppose que chaque ligne de cette matrice et triée, mais aussi que chaque colonne de la matrice est triée. Le but est de rechercher si une

valeur se trouve dans la matrice, avec une fonction de type `'a array array -> 'a -> bool`.

▶ **Question 4** Écrire une fonction naïve `recherche_naive` de complexité en $\mathcal{O}(np)$.

▶ **Question 5** Proposer une variante de complexité $\mathcal{O}(n \log_2 p)$. Quelle amélioration peut-on apporter à cet algorithme si l'on sait que p est bien plus petit que n (par exemple, $p^2 = n$)?

▶ **Question 6** Proposer une variante de complexite $\mathcal{O}(n + p)$, en justifiant sa correction sommairement (par exemple, en proposant un invariant de boucle).

▶ **Question 7** Supposons que $n = p$: montrer qu'il est impossible de résoudre ce problème en moins de n comparaisons. Que peut-on en conclure sur la fonction précédente?

▶ **Question 8** Écrire une fonction `un_de_perdu : int array -> int` qui prend en entrée un tableau de taille n contenant tous les entiers de $[\![0, n]\!]$, sauf un, et qui renvoie cet entier manquant. Par exemple, on aura :

OCaml
```
# un_de_perdu [|5; 2; 4; 1; 0|];;
- : int = 3
```

Votre solution devra avoir une complexité temporelle linéaire et spatiale constante et marcher jusqu'à $n = 50000$.