TP : Les vecteurs – Mise en pratique

A rendre par email au plus tard le dimanche 11 novembre

1) Références et boucles `while`

1.a) Références : la théorie

Les variables permettent de stocker des valeurs, mais comme on l'a vu en cours, ces contenus sont figés : il n'est pas possible de modifier la valeur d'une variable une fois qu'on l'a déclarée.

Lorsqu'on veut pouvoir modifier le contenu d'un élément que l'on stocke, on utilise une **référence**.

La syntaxe est un peu particulière :

- Pour déclarer une référence, on utilise le mot clef `ref` après le signe égal :
  `let [nom] = ref [valeur] ;;`
  Exemple : `let x = ref 3 ;;`

- Pour obtenir la valeur actuelle de cette référence, on utilise un point d'exclamation `!` :
  `![valeur] ;;`
  Exemple : `!x;;`

- Pour modifier cette valeur, on utilise le "deux points, égal" `:=`
  `nom := [nom_valeur];;`
  Exemple : `x := 4;;`

1.b) Références : la pratique

Appropriez-vous les notations précédentes en créant et en modifiant des références d'au moins deux types différents (par exemple `float` et `string`).

1.c) Remarque : déclarations à l'intérieur d'une fonction

Jusque-là, nous avons vu des syntaxes de la forme `let [nom] = [valeur];;`

Néanmoins, si vous devez déclarer une variable ou une référence à l'intérieur d'une fonction, utilisez plutôt la syntaxe `let [nom] = [valeur] in` (on a remplacé les double points-virgules par le mot clef `in`).

Un schéma classique est donc le suivant

```
let [nom_fonction] [args] =
      [debut_fonction]
      let [nom_variable] = [valeur_variable] in
      [suite_fonction];;
```

1.d) Boucles `while`

Les références ne sont pas forcément faciles à prendre en main, mais elles se révèlent très pratiques dans le cadre d'une boucle `white`.

Comme on l'a vu en cours, et comme l'indique la syntaxe, l'idée est de répéter une série d'instructions tant qu'une condition est vérifiée :

```
while (cond)
do
      instr1;
      instr2;
      …
      instrN-1;
      instrN      (La dernière instruction ne prend pas de point-virgule)
done;;
```

Pour commencer, écrivez une série d'instructions qui affiche grâce à une boucle `while` tous les nombres de 1 à 10.

Modifiez ensuite votre code pour obtenir une fonction qui prend en argument un entier `n` et affiche, toujours grâce à une boucle `while`, tous les nombres de 1 à `n`.

1.e) Tirages aléatoires

De nombreux problèmes informatiques supposent qu'on dispose d'un générateur aléatoire, c'est-à-dire d'un outil capable de tirer des nombres au hasard.

En Caml, on utilise pour cela les lignes de code suivantes :

- `Random.self_init();;`

Exécutez cette ligne de code au début de votre fichier afin de "démarrer" le générateur aléatoire.

- `Random.int [borne_superieure]`

La fonction `Random.int` fonction prend un argument n et renvoie un nombre entier tiré au hasard entre 0 (inclus) et n (exclus).

Pour vous familiariser avec ces outils, imaginez une fonction `pile_ou_face` qui ne prend aucun argument et qui renvoie soit "pile", soit "face" (avec la même probabilité pour chaque).

Ecrivez ensuite une fonction `de_a_six_faces` qui renvoie un entier tiré au hasard entre 1 et 6.

Combinez ces techniques de tirage aléatoire avec une boucle `while` pour écrire une fonction `essaye_encore` qui :

- Tire un nombre au hasard entre 0 et 10
- Affiche ce nombre
- Recommence si ce nombre n'est pas égal à 10.

Comment pourriez-vous modifier cette fonction pour compter le nombre d'essai nécessaire avant d'obtenir un 10 ?

★ Sauriez-vous en outre modifier encore cette fonction afin que la valeur à atteindre (jusque-là fixée à 10) soit elle aussi tirée au hasard avant d'entrer dans la boucle `while` ?

2) Fonctions récursives

2.a) Syntaxe :

Pour définir une fonction récursive en Caml, on utilise le mot clef `rec` :

```
let rec [nom_fonction] [args] =
      [debut_fonction]
      [nom_fonctions] [args_bis]
      [suite_fonction];;
```

Attention à ne pas oublier le cas de base, sinon votre programme risque de boucler à l'infini (pour plus de sécurité, sauvegardez régulièrement votre code).

2.b) Premier exemple : la factorielle

Implémentez en Caml la fonction `factorielle` d'après le pseudo-code suivant :

```
Factorielle n =
      Si n = 0
            Alors 1
            Sinon n * Factorielle(n-1)
      Fin si
```

2.c) Suite de Fibonacci :

On définit la suite de Fibonacci de la manière suivante :

$$Fib(n) = \begin{cases} 0 \, si \, n = 0 \\ 1 \, si \, n = 1 \\ Fib(n-2) + Fib(n-1) si \, n > 1 \end{cases}$$

Ecrivez la fonction correspondante (vous aurez besoin de deux `if…then…else`).

★ Selon vous, peut-on écrire ce programme de façon plus intelligente ? Est-ce qu'on ne fait pas plusieurs fois les mêmes calculs ?

Imaginez (et si possible tapez) une nouvelle fonction permet de calculer *Fib(n)* en n étapes.

3) A la découverte des vecteurs

Commencez par définir quelques vecteurs qui nous servirons pour les tests (observez au passage les types des objets ainsi définis :

```
let v1 = [| 23 ; 42 ; 51 |];;
let v2 = Array.make 3 "true";;
let v3 = Array.make 4 3.14 ;;
let v4 = Array.init 5 (function x->2*x);;
```

Ecrivez ensuite les fonctions suivantes :

- Affichage des différents éléments d'un vecteur d'entiers
  `print_int_array : int array -> unit`

- Calcul de la valeur moyenne d'un vecteur de nombres réels
  `average_float_array : float array -> float`

- Test "Ce vecteur est-il trié par ordre croissant ?"
  `est_croissant : 'a array -> bool`

Remarque : Il n'y a pas d'erreur dans le type proposé pour la dernière fonction : si vous l'avez correctement implémentée, vous avez sans le savoir programmé une fonction *polymorphe*, c'est-à-dire utilisable avec des objets de types différents. Essayez !

4) Recherche dans un vecteur non trié

Dans un premier temps, on suppose qu'on travaille avec des vecteurs non triés.

Ecrivez et testez les fonctions suivantes :

- Avec une boucle for
  `appartient_for : 'a array -> 'a -> bool`

- Avec une boucle while
  `appartient_while : 'a array -> 'a -> float`

5) Exponentiation rapide

L'objectif de cette partie est d'implémenter une méthode type "Diviser pour régner", autour de l'exemple de l'algorithme d'exponentiation rapide

4.a) Exponentiation classique

Commencez par écrire deux fonctions qui calculent "un peu bêtement" a^n à partir de la formule $a^n = a \times a \times \cdots \times a$:

- une fonction itérative (avec une boucle `for`)
- une fonction récursive

4.b) Exponentiation rapide

On a vu en cours qu'on pouvait faire ça plus rapidement en s'inspirant de la formule suivante

$$x^n = \begin{cases} x & si\ n = 1 \\ (x^2)^{n/2} & si\ n\ est\ pair \\ x \times (x^2)^{(n-1)/2} & si\ n\ est\ impair \end{cases}$$

Implémentez une nouvelle fonction à partir cette formule.

Comparez vos résultats avec ceux obtenus avec les versions précédentes. Attention à ne pas prendre des nombres trop grands ou vous aurez des surprises (pourquoi ?).

4.c) Compter le nombre de multiplication

Pour réaliser les avantages de la méthode "Diviser pour régner" sur la méthode classique, il est intéressant de comparer le nombre de multiplications effectuées avec chaque méthode.

★ Essayez d'imaginer (et d'implémenter) de nouvelles fonctions qui renvoient en plus du résultat final le nombre de multiplications nécessaires pour obtenir ce dernier.

Indice 1 : Pensez récursif.

Indice 2 : Si la fonction récursive `exp_rapide_2 : int -> int -> (int * int)` est telle que `exp_rapide_2 x n` renvoie `(y , nb_m)` où `y` est le résultat attendu (x^n) et `nb_m` est le nombre de multiplication nécessaires pour le calcul, alors on pourra faire des appels récursifs en écrivant par exemple

```
let (y, nb_m) = exp_rapide_2 (x*x) (n/2)    (* Appel récursif *)
in
(y, nb_m + 1)    (* Solution renvoyée *)
```

6) Recherche dans un vecteur trié

5.a) Un problème plus général

On suppose désormais que le vecteur qu'on manipule est trié. Comme on l'a vu en cours, on peut dans ce cas utiliser une approche "Diviser pour régner" pour tester si un élément appartient à un vecteur.

En vous aidant si besoin est du support de cours en ligne, écrivez pour commencer une fonction `appartient1 : 'a array -> 'a -> int -> int -> bool` qui teste si un élément `x` apparaît dans un vecteur `v` entre les indices `a` et `b`.

5.b) Encapsulation

Comme en pratique on travaille presque toujours avec `a=0` et `b=Array.length(v)-1`, on utilise le procédé dit d' "encapsulation" pour définir une fonction auxiliaire à l'intérieur de la fonction destinée à l'utilisateur.

Implémentez une nouvelle fonction `appartient : 'a array -> 'a -> bool` qui ne prend que deux arguments : le vecteur `v` et l'élément `x` recherché.

5.c) Comparaisons des deux méthodes

★ Ecrivez une nouvelle fonction qui, en plus de déterminer si un élément `x` appartient à un vecteur `v` (qu'on suppose trié), renvoie le nombre de comparaisons nécessaires pour arriver à cette conclusion.

★ Modifiez également la fonction de recherche classique (sur un vecteur non trié) afin d'obtenir également le nombre de comparaisons nécessaires avec cette méthode.

★ Comparez les performances des deux approches sur différents vecteurs triés : que se passe-t-il quand l'élément est au début du vecteur ? à la fin ? Et quand il n'apparait pas du tout ?