Bases d'OCaml, suite

I. Récursion et itération

1. Algorithme d'Euclide

On rappelle que le PGCD de deux entiers a et b peut se calculer en utilisant les relations :

$$\text{PGCD}(a,b) = \begin{cases} a & \text{si } b = 0 \\ \text{PGCD}(b, a \bmod b) & \text{sinon} \end{cases}$$

Question 1 Écrire une version itérative du calcul de PGCD, à base de boucles et de références.

Question 2 Écrire une version récursive de la même fonction.

2. Quelques algorithmes de tri

2.1. Tri par insertion sur des tableaux

On rappelle que le tri par insertion procède de la façon suivante : Supposons que les k premières valeurs du tableau sont triées (par exemple, pour $k = 4$, les premières valeurs triées étant représentées en gras) :

```
[2, 6, 7, 9, 3, 4, 1, 8, 0, 5]
```

On désire maintenant que les $k+1$ premières valeurs soient triées. Pour cela, il faut mettre la $k+1$-ème valeur (de position initiale k) à la bonne place par rapport aux valeurs de plus petit incide.

En notant v la valeur en cours de traitement, on commence par regarder la valeur en position $k-1$ (donc la position juste à gauche de la valeur à classer, qui correspond à la plus grande des valeurs déjà classées) et si celle-ci est plus grand que v, on les échange et recommence jusqu'à trouver une valeur strictement plus petite que v (ou arriver au début du tableau).

Voici par exemple les différentes étapes conduisant à l'insertion de $v = 3$ (initialement en position 4) à sa bonne place :

```
[2, 6, 7, 9, 3, 4, 1, 8, 0, 5]
[2, 6, 7, 3, 9, 4, 1, 8, 0, 5]
[2, 6, 3, 7, 9, 4, 1, 8, 0, 5]
[2, 3, 6, 7, 9, 4, 1, 8, 0, 5]
[2, 3, 6, 7, 9, 4, 1, 8, 0, 5]
```

Question 3 Écrire une fonction `insertion_tableau` telle que `insertion t p` effectué à la bonne place de la valeur située en position initiale p dans le tableau t, dont on supposera que les p premières valeurs (d'indices 0 à $p-1$) sont triées par ordre croissant. Cette fonction ne renvoie rien, mais modifie le tableau t.

Question 4 Écrire la fonction `tri_insertion_tableau` qui, étant donné un tableau t, va le trier en le parcourant de gauche à droite et en effectuant l'insertion successive des différentes valeurs. À nouveau, cette fonction ne renvoie rien mais le tableau passé en argument doit avoir à la fin son contenu trié.

2.2. Tri par insertion sur des listes

On peut très facilement adapter le tri par insertion aux listes.

Question 5 Écrire une fonction `insertion_liste` qui, étant donné un élément et une liste supposée triée, retourne la liste obtenue en ajoutant le nouvel élément à la bonne place.

On veut, par exemple :

```
# insertion_liste 4 [1; 3; 5; 6; 8] ;;
- : int list = [1; 3; 4; 5; 6; 8]
```

Question 6 En déduire une fonction `tri_insertion_liste` qui effectue le tri par insertion de la liste passée en argument.

2.3. Tri fusion

Nous allons maintenant adapter le tri fusion à OCaml de façon totale fonctionnelle, pour trier des listes et non plus des tableaux.

Pour pouvoir tester l'algorithme que vous allez écrire, il faut s'assurer que la liste renvoyée est bien triée.

Question 7 Écrire une fonction `est_croissante_liste : 'a list -> bool` qui indique si la liste passée en argument est croissante.

Question 8

1. En fait, écrire aussi une fonction

```
est_croissante_tableau : 'a array -> bool
```

   qui indique si le tableau passé en argument est croissant.

2. Tester la fonction `tri_insertion_tableau` précédente.

Revenons maintenant à

Question 9 Écrire une fonction `fusion` qui prends en entrée deux listes triées de même type, et retourne la liste correspondant à leur fusion ordonnées.

Par exemple, on veut :

```
# fusion [1; 3; 5; 6; 8] [2; 3; 4; 5; 7] ;;
- : int list = [1; 2; 3; 3; 4; 5; 5; 6; 7; 8]
```

Question 10 Écrire une fonction `division` qui, étant donné une liste, retourne un couple de listes correspondant chacune à une moitié de la liste initiale. On peut avoir, par exemple :

```
# division [1; 2; 3; 4; 5; 6; 7; 8; 9] ;;
- : int list * int list = ([1; 3; 5; 7; 9], [2; 4; 6; 8])
```

Question 11 En déduire une fonction `tri_fusion : 'a list -> 'a list` qui prends en entrée une liste et en renvoie une version triée.

II. Jouons avec les types sommes

1. Logique ternaire

```
type ternaire =
  | Vrai
  | Faux
  | Inconnu
```

Question 12 Écrire une fonction

```
negation : ternaire -> ternaire
```

qui transforme `Vrai` en `Faux` et vice-versa, et laisse `Inconnu` inchangé.

Question 13 Écrire une fonction

```
conjonction : ternaire -> ternaire -> ternaire
```

qui renvoie `Vrai` si les deux arguments sont `Vrai`, `Faux` si l'un des deux arguments est `Faux`, et `Inconnu` sinon.

2. Autour du type option

Il existe en OCaml un type `'a option` défini de la façon suivante (ne recopiez pas ce qui suit, le type est défini par défaut) :

```
type 'a option = None | Some of 'a
```

Une utilité, par exemple, est lorsque l'on utilise avec une *liste d'association* qui est une liste de couples *clé/valeur* et qui, pour renvoie la valeur associée si elle existe (avec un `Some`), et `None` sinon.

```
# List.assoc_opt 2 [ (1, "One"); (2, "Two"); (3, "Three") ] ;;
- : string option = Some "Two"
# List.assoc_opt 4 [ (1, "One"); (2, "Two"); (3, "Three") ] ;;
- : string option = None
```

Question 14 Écrire la fonction

```
assoc_opt : 'a -> ('a * 'b) list -> 'b option
```

dont on vient de décrire le comportement (même si elle existe déjà dans la bibliothèque `List`).

Question 15 Écrire une fonction

```
premiere_occurence : 'a array -> 'a -> int option
```

qui, étant donné un tableau t et une valeur a, renvoie `Some` p où p est la position de la première occurence de a dans t si celle-ci est définie, ou bien `None` sinon.

On veut, par exemple,

```
# premiere_occurence [| 1; 2; 3; 4; 2; 3; 3; 4 |] 4 ;;
- : int option = Some 3
# premiere_occurence [| 1; 2; 3; 4; 2; 3; 3; 4 |] 5 ;;
- : int option = None
```

Question 16 Écrire une fonction

```
appli_option : ('a -> 'b) -> 'a option -> 'b option
```

ayant le comportement suivant :

```
# appli_option (fun x -> x * x) (Some 2) ;;
- : int option = Some 4
# appli_option (fun x -> x * x) None ;;
- : int option = None
```

Question 17 Écrire une fonction

```
filtre_option : 'a option list -> 'a list
```

ayant le comportement suivant :

```
# filtre_option [ Some 1; None; None; Some 2; Some 3; None] ;;
- : int list = [1; 2; 3]
```

3. Manipulation d'expressions

On considère le type suivant :

```
type expr =
  | Const of int
  | Var of string
  | Add of expr * expr
  | Sub of expr * expr
  | Mult of expr * expr
  | Div of expr * expr
```

qui va nous permettre de manipuler des expressions arithmétiques simples.

Question 18 Écrire une fonction `subst` qui, étant donné une expression e_1, une chaîne de caractères `v` et une expression e_2, remplace chaque occurence de `Var v` dans e_1 par e_2.

Question 19 Écrire une fonction `eval : expr -> expr` qui évalue autant que possible l'expression passée en argument. En particulier, si l'expression ne contient aucune variable, l'expression renvoyée doit être de constructeur `Const`.

Question 20 Écrire une fonction `simpl : expr -> expr` qui simplifie, autant que possible, l'expression passée en argument.

Question 21 Étendre la définition du type pour prendre en compte les flottants en plus des entiers.

Question 22 Étendre le type et les fonctions précédentes pour prendre en compte des fonctions simples (comme par exemple ln, exp, cos et sin).

Question 23 Écrire une fonction de dérivation d'expression selon une variable.