TP9: Listes, récursivité et types somme

MP2I Lycée Pierre de Fermat

Ce TP est à rendre avant le samedi 4 février à 22h sur Cahier de Prépa.

Vous rendrez un fichier ".ml" par exercice, ainsi qu'un petit compte rendu écrit pour d'éventuelles réponses / remarques. Pensez à bien commenter vos fonctions, en précisant :

— Une description des paramètres

— Les éventuelles hypothèses sur les arguments

— La valeur renvoyée en fonction des arguments

Par exemple :

```
(* Renvoie true si x divise y. x et y doivent
   être des enviers positifs, et y doit être
   non-nul. *)
let divise x y =
  if y <= 0 then failwith "Division par zéro"
  else match (x mod y) with
    | 0 -> true
    | _ -> false;;
```

Listes

On rappelle la syntaxe des listes :

— La liste vide s'écrit `[]`, elle est de type `'a list`

— Si `E1` est une expression de type `'a` et `E2` une expression de type `'a list`, alors `E1 :: E2` est aussi de type `'a list`, et contient E1 comme premier élément, suivi des éléments de E2.

— On peut utiliser `[]` et `::` dans les motifs :

```
let rec longueur l = match l with
  | [] -> 0
  | x :: q -> 1 + longueur q ;;
```

Exercice 1. Étudions quelques fonctions sur les listes. Pour chacune des fonctions que vous écrivez, donnez également un jeu de test pertinent (cas limites, couverture du code...). Tentez de réutiliser au maximum les fonctions que vous définissez.

Question 1. Écrivez une fonction `somme: int list -> int` renvoyant la somme d'une liste d'entiers.

Question 2. Écrivez une fonction `recherche: 'a list -> 'a -> bool` qui, étant donné L une liste et x un élément, détermine si $x \in L$.

Question 3. Écrivez une fonction `max: 'a list -> 'a` renvoyant le maximum d'une liste non-vide.

Question 4. Pour deux listes $L_1 = [x_1; ...; x_n]$ et $L_2 = [y_1; ...; y_m]$, la concaténation de L_1 et L_2 est $L = [x_1; \dots; x_n; y_1; \dots; y_m]$. Écrivez une fonction `concatener: 'a list -> 'a list -> 'a list` qui prend en entrée deux listes et renvoie leur concaténation.

Question 5. Écrivez une fonction `multi_concat: 'a list list -> 'a list` prenant en entrée une liste de listes et renvoyant leur concaténation. Faites bien attention à l'ordre : testez votre fonction !

Question 6. Écrivez une fonction `map: ('a -> 'b)-> 'a list -> 'b list` prenant en entrée une liste L, une fonction f, et renvoyant la liste obtenue en appliquant f à chaque élément de L.

Question 7. Écrivez une fonction `range: int -> int list` prenant en entrée un entier n et renvoyant la liste $[0; 1; \dots; n-1]$. Vérifiez que votre fonction ne renvoie pas la liste dans l'ordre inverse.

Question 8. Écrivez une fonction `make_list: (int -> 'a)-> int -> 'a list` prenant en entrée une fonction f, un entier n, et renvoyant la liste $[f(0); f(1); \dots; f(n-1)]$.

Question 9. Écrivez une fonction `compose_liste` prenant en entrée une liste de fonctions et renvoyant la composée de toutes les fonctions. Avant de coder, réfléchissez au sens précis dans lequel les compositions auront lieu, et donc au type précis qu'aura votre fonction. Vous documenterez la fonction avec un commentaire expliquant clairement son comportement.

Pour $k \in \mathbb{N}$, on note $f_k \in \mathbb{N}^{\mathbb{N}}$ la fonction suivante :

$$f_k : \begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N} \\ x & \longmapsto & \begin{cases} \frac{x}{2+k} & \text{si } x = 0 \text{ modulo } 2+k \\ 3x+k & \text{sinon} \end{cases} \end{array}$$

On note ensuite $F_n = f_0 \circ f_1 \cdots \circ f_{n-1}$ et $F : n \mapsto F_n(0) + \cdots + F_n(n-1)$

Question 10. Définir une fonction OCaml calculant les f_k, puis une fonction calculant F. Vous pouvez obtenir F uniquement en utilisant les fonctions définies plus haut. Vérifiez :

- $F(3) = 83$
- $F(20) = 60321545051$
- $F(30) = 12254504478085045$

Exercice 2.

Question 1. Reprenez le code des fonctions `recherche`, `somme` et `multi_concat`. Que constatez-vous ?

On se propose d'écrire une fonction qui permettrait de généraliser ces trois fonctions. Cette fonction, très classique en programmation fonctionnelle, s'appelle ***fold*** ou ***reduce*** :

```
let rec fold f a l = match l with
| [] -> a
| x::q -> f x (fold f a q) ;;
```

Question 2. Recopiez la fonction fold, et évaluez :

```
fold (fun x y -> x^y) "" ["vive"; " "; "OCaml"; "!!!"];;
(* les opérateurs sont des fonctions, on peut donc également écrire: *)
fold (^) "" ["vive"; " "; "OCaml"; "!!!"];;
```

`fold` sert donc à utiliser les éléments d'une liste pour accumuler un résultat à partir d'un élément de départ a et d'une fonction d'agrégation f.

Question 3. En utilisant fold, donnez une nouvelle définition des fonctions `somme`, `recherche` et `multi_concat`.

Deux autres fonctions très utilisées en OCaml sont `map: ('a -> 'b)-> 'a list -> 'b list` (vue plus haut) et `filter: 'a list -> ('a -> bool)-> 'a list` qui prend en entrée une liste et une fonction de filtre, et renvoie la liste des éléments qui passent le filtre.

Question 4. Définissez `filter` et `map`, d'abord directement, puis en utilisant `reduce` sur des fonctions bien choisies

Intéressons nous à la manipulation des strings. Pour accéder au k-ème caractère d'un string s, on utilise la syntaxe `s.[k]` (avec $0 \leq k < |s|$). En OCaml, on peut accéder aux fonctions concernant les strings avec `String.bla`. Par exemple, `String.length` est la fonction qui calcule la longueur d'un string.

Question 5. Écrire une fonction `list_of_string: string -> char list` permettant de décomposer un string en liste de caractères.

On admet que `String.of_seq (List.to_seq) l` permet de retransformer une liste `l: (char list)` en un string.

Question 6. Écrire une fonction `split: string -> char -> char list list` permettant de diviser un string en mots, selon un caractère de séparation. Par exemple :

```
assert (split ',' "toto,tata,tutu" = ["toto"; "tata"; "tutu"]) ;;
```

Il pourra être utile d'utiliser une fonction auxiliaire de la forme :

```
let rec split_from_i (s:string) (sep:char) (i:int) (curr:char list) =
  ...
```

qui permet de diviser s en liste de strings, à partir de l'indice i, en ayant déjà lu les caractères dans $curr$.

Question 7. Écrire une fonction de décomposition en base B : `decomp: int -> string -> (int*int)` pour $B \in [\![2, 10]\!]$ (i.e. avec uniquement des chiffres classiques). Cette fonction renverra l'entier lu ainsi que l'indice du premier caractère ayant empêché la lecture, soit parce que l'on est à la fin du string, soit parce qu'on lit autre chose qu'un chiffre. Par exemple :

```
assert (decomp 10 "56+32" = (56,2)) ;;
```

Vous pourrez utiliser la fonction `Char.code` qui donne le code ASCII d'un caractère.

Types sommes

En OCaml, on peut définir des types customisés, appelés les ***types sommes***. Un type somme représente une structure, un objet, une situation, composé de plusieurs cas. Par exemple :

```
type couleur = Coeur | Pique | Carreau | Trefle ;;
```

Cette syntaxe signifie que l'on a un type appelé `Couleur`, contenant 4 valeurs. On peut utiliser ces valeurs dans des expressions et dans les motifs :

```
let couleur1 = Pique ;;
let est_rouge c = match c with
  | Coeur   -> true
  | Carreau -> true
  | Pique   -> false
  | Trefle  -> false ;;

assert (est_rouge Carreau);; (* vaut true *)
```

Un type somme peut également contenir des valeurs à paramètres. Par exemple :

```
type carte =
  | Nombre of (int * couleur) (* Cartes 2 à 10 *)
  | Valet of couleur
  | Dame of couleur
  | Roi of couleur
  | As of couleur ;;

let carte_1 = Valet Coeur ;;
let carte_2 = Nombre (9, Pique) ;;

(* Renvoie la couleur d'une carte *)
let couleur_de_carte ca = match ca with
  | Nombre (n, cou) -> cou
  | Valet cou -> cou
  ...
  | As cou -> cou ;;
assert (couleur_de_carte (Roi Trefle) = Trefle) ;;
```

Les mots `Nombre`, ... `As` sont appelés des ***constructeurs***. Ce ne sont pas des fonctions ! Par ailleurs, les constructeurs d'un type doivent forcément commencer par une lettre majuscule, et le nom d'un type doit commencer par une lettre minuscule.

Exercice 3.

Téléchargez le fichier "cartes.ml" sur cahier de prépa, qui contient les types `couleur` et `carte`, ainsi que les fonctions associées

Question 1. Écrivez une fonction `string_of_couleur: couleur -> string` qui renvoie le nom d'une couleur sous forme de chaîne de caractère

Question 2. Écrivez une fonction `string_of_carte: carte -> string` qui renvoie le nom d'une carte sous forme de chaîne de caractère : "Dame de pique", "10 de coeur", etc...

On représente une main ou un deck de cartes par une liste de cartes : `carte list`.

Question 3. Tentez de comparer quelques cartes avec <, <=, =, etc... que remarquez vous ?

Question 4. Écrivez une fonction qui permet d'insérer une carte au bon endroit dans une main, de sorte à avoir tous les 2, puis tous les 3, et ainsi de suite jusqu'aux as. Au sein d'une même valeur, les cartes seront triée dans l'ordre coeur-carreau-pique-trfle.

Question 5. Écrivez une fonction qui permet d'insérer une carte au bon endroit dans une main, sachant que l'on veut regrouper les cartes par couleur, puis au sein d'une même couleur les classer par ordre. Il pourra être utile de coder une fonction permettant de comparer les cartes selon cet ordre.

Question 6. Utilisez cette fonction pour écrire une fonction implémentant le tri par insertion et permettant de trier une main de cartes

Question 7. Écrivez une fonction `gen_couleur` qui prend en entrée une couleur et renvoie la liste des 13 cartes de cette couleur, dans un ordre quelconque.

Question 8. A l'aide des deux fonctions précédentes, écrivez une fonction de signature `deal: carte list -> carte list * carte list` qui distribue un deck entre deux joueurs, en alternant, et en distribuant l'intégralité des cartes.

Exercice 4.

On veut créer un type permettant de représenter les boissons. On propose d'avoir les boissons suivantes :

- De l'eau
- Du jus de fruit (il faut préciser quel fruit)
- Du Breizh Cola,qui peut être normal ou light

Question 1. Créer un type `type boisson = ... ;;` permettant de représenter les boissons.

Question 2. Créer une fonction qui calcule le prix au litre d'une boisson. On pose :

- L'eau est gratuite
- Tous les jus coûtent 3€ le litre, sauf le jus de ramboutan qui coûte 5.30€ le litre
- Le Breizh Cola coûte 1€ le litre

Rien n'emêche un type d'être récursif, c'est à dire d'avoir un constructeur utilisant le type lui-même.

Question 3. Modifier le type `boisson` en lui ajoutant un constructeur `Cocktail` permettant de représenter des mélanges. On veut pouvoir mélanger deux boissons B_1, B_2 en précisant la fraction de B_1.

Question 4. Modifier la boisson de calcul de prix pour prendre en compte ce nouveau constructeur.

Question 5. Créer une fonction `shaker: boisson list -> boisson` prenant en entrée une liste non-vide de boissons $B_1 ... B_n$ et faisant un gigantesque cocktail, de la forme :

$$\textbf{cocktail}(\frac{1}{2}, B_1, \textbf{cocktail}(\frac{1}{2}, B_2, \textbf{cocktail}(...\textbf{cocktail}(\frac{1}{2}, B_{n-1}, B_n)...)$$

On voudrait pouvoir afficher la recette d'un cocktail [1], sous la forme :

```
Recette pour 1L:
50 mL Eau
400 mL Jus de raisin
300 mL Breizh Cola
250 mL Jus d'orange
```

1. Ne pas reproduire le cocktail donné en exemple chez vous

Question 6. Écrire une fonction `string_of_boisson` calculant le nom d'une boisson hors-cocktail.

Question 7. Écrire une fonction `ingredients: boisson -> (boisson*float) list` permettant de transformer une boisson en une liste de couples (B, x) où B est une boisson de base, et x la proportion de cette boisson dans le cocktail. On s'autorisera à avoir des doublons, par exemple :

```
1 ingredients (Cocktail(Eau, Eau, 0.5));;
2 (* Vaut [(Eau, 0.5); (Eau, 0.5)] *)
```

Question 8. Écrire une fonction permettant d'éliminer les doublons en ajoutant les proportions, et en déduire une fonction permetttant d'afficher la recette d'une boisson comme demandé.

En bonus

Exercice 5.

Reprenons le tri rapide, que nous avons déjà vu en TP en C :

Algorithme 1 : Tri rapide

Entrée(s) : L une liste d'éléments
Sortie(s) : L' liste triée des éléments de L

si L ***est de taille 0 ou 1*** **alors**
  retourner L
$p \leftarrow$ la tête de L ;
$L_{\leq} \leftarrow$ Liste des éléments $y \in L$ tels que $y \leq p$;
$L_{>} \leftarrow$ Liste des éléments $y \in L$ tels que $y > p$;
$L' \leftarrow$ la concaténation de $L_{\leq}$ et $L_{>}$;
retourner *L'*

Question 1. Écrivez une fonction `separer: 'a -> 'a list -> ('a list * 'a list)` prenant en entrée un élément x et une liste L et renvoyant $L_{\leq}, L_{>}$ définies comme dans l'algorithme plus haut.

Question 2. Écrivez une fonction `tri_rapide: 'a list -> 'a list` triant sa liste d'entrée. Donnez également un jeu de test pour vérifier votre fonction.

Question 3. Modifiez les deux fonctions précédentes pour qu'elles prennent également en paramètre une fonction de comparaison, de façon à pouvoir trier selon un autre ordre que celui par défaut. La fonction tri_rapide devra être de la forme :

```
1 let rec tri_rapide (l: 'a list) (comp:('a -> 'a -> bool)) = ... ;;
```

où `comp x y` renvoie vrai si $x < y$ selon l'ordre choisi, faux sinon.

Question 4. Utilisez cette fonction pour trier un deck de carte