TP10: Récursivité, listes, types somme
Rapport

MP2I Lycée Pierre de Fermat

Remarques générales

Commentaires Il faut commenter les fonctions, en respectant les règles qu'on a déjà évoqué des dizaines de fois ! Par exemple, le nom de chaque paramètre doit apparaître dans la description. Le commentaire suivant est correct :

```
(* Nombre d'éléments de l *)
let rec longueur (l: 'a list) : int = ...
```

Le commentaire suivant ne l'est pas :

```
(* Renvoie le nombre d'éléments d'une liste *)
let rec longueur (l: 'a list) : int = ...
```

Lorsque l'on code des fonctions assez mathématiques,comme la fonction `map`, on peut tout à fait les documenter en passant par des formules/équations :

```
(* Applique f sur chaque élément de l:
   map f [x1; x2; ...; xk] = [f x1; f x2; ...; f xk] *)
let rec map (f: 'a -> 'b) (l: 'a list) : 'b list = ..
```

Pour voir plus d'exemples, vous pouvez regarder la documentation officielle OCaml du module List : ocaml.org/manual/5.3/api/List.html

N'oubliez pas également de typer vos fonctions !

Doubles points-virgules et points-virgules Il reste quelques rendus dans lesquels il y a des doubles points-virgules inutiles. Pour rappel, il ne faut pas en mettre à la fin d'un let, mais à la fin d'une **suite** de let ou à la fin d'une **expression** :

```
let f x y = ....

let g a = ....

let u = .... ;; (* fin d'une suite de let, suivie d'une expression *)

g (f 2 u);; (* fin d'une expression *)
g (f 3 4);; (* fin d'une expression *)

let ...
```

Il faut aussi bien faire la distinction entre `;;` et `;`, qui sert à séparer des instructions. Par exemple, le code suivant n'est pas correct :

```
let test () =
  assert truc ;;
  assert bidule ;;
  ...
  assert machin ;;
```

La fonction `test` ne contiendra que la première assertion !! En effet, après le premier `;;`, la définition de la fonction prend fin. Les assertions suivantes sont exécutées directement lorsque l'on exécute le code dans utop, au lieu d'attendre que l'on lance la fonction `test ()`. Pour corriger, on écrira :

```
let test () =
  assert truc ; (* séparer les instructions de la fonction test *)
  assert bidule ; (* idem *)
  ...
  assert machin
;; (* fin de la définition de test () *)
test () ;; (* lancer les tests *)
```

Toujours sur le sujet des points-virgules : ils servent à **séparer** deux instructions, pas à marquer la fin comme en C. Pendant les séances de TP, vous avez pu rencontrer des erreurs étranges où utop vous dit qu'il y a une erreur de syntaxe à la toute fin de votre fichier, sans raison apparente. C'était quasi-systématiquement à cause d'un point virgule mis sans rien à la suite !
Par exemple, si l'on regarde le code suivant :

```
let test1 () =
  assert bla;
  assert truc;

let test 2 () =
  assert machin
```

Le point virgule à la fin de `test1` attend une instruction à sa suite. Le `let test2 () = ...` est donc compris comme faisant partie de la définition de `test1` : OCaml pense que c'est un `let` local, attendant un `in`. Comme il atteint la fin du fichier sans voir de `in`, il indique une erreur de syntaxe. Pour avoir une version correcte, on écrira :

```
let test1 () =
  assert bla;
  assert truc (* pas de point virgule ! *)

let test 2 () =
  assert machin
```

Conclusion : mettre un point virgule sans rien à la suite, c'est comme écrire `1+2+` !

Parenthèses Une difficulté en OCaml est de s'habituer au fait que les parenthèses ne servent pas à marquer les applications de fonctions, mais à faire des blocs. Pendant les séances, j'ai souvent vu :

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

Ce n'est pas correct : la dernière paire de parenthèses, autour de `f q`, ne doit pas être là. L'expression est comprise comme "J'applique f à q, et ensuite j'applique map au résultat", alors que l'on veut plutôt dire "J'applique map à f et à q". La version correcte est donc simplement `map f q`.

De même, lorsque l'on fait des applications à la chaîne, il **faut** mettre certaines parenthèses. Par exemple :

```
let rec multi_concat ll = match ll with
  | [] -> []
  | x::q -> concatener x (multi_concat q)
```

Ici, les parenthèses servent, car le deuxième argument de `concatener` est `multi_concat q`. Si l'on enlève les parenthèses :

```
concatener x multi_concat q
```

c'est une erreur car OCaml croit que l'on appelle `concatener` avec trois arguments alors qu'elle n'en prend que deux.

Exercice 1

Il est utile de connaître l'ordre de priorité des différents éléments d'OCaml (opérateurs, application de fonction, etc...). Ceci permet d'enlever les parenthèses inutiles. Par exemple, l'application de fonction est prioritaire sur le constructeur de listes `::`, donc on peut écrire :

```
let rec map (f: 'a -> 'b) (l: 'a list) : 'b list = match l with
    | [] -> []
    | x::q -> f x :: map f q
```

sans parenthèses dans la dernière ligne.

Il était possible de faire la dernière question avec une fonction auxiliaire différente de celle demandée sur le sujet, en rajoutant un paramètre permettant de fixer le départ de la liste :

```
(* range_between a b = [a; a+1; ...; b-1; b] *)
let rec range_between (a: int) (b: int) : int list =
  if a > b then []
  else a :: range_between (a+1) b
let range n = range_between 0 (n-1)
```

Notons qu'elle n'est pas récursive terminale, contrairement à celle vers laquelle vous pousse l'énoncé :

```
(* range_concat n [x1; ... xk] = [0; 1; ...; n-1; x1; ... xk]  *)
let rec range_concat (n: int) (accu: int list) : int list =
  if n = 0 then accu
  else range_concat (n-1) ((n-1) :: accu)
let range n = range_concat n []
```

Exercice 2

Cet exercice était un peu dur mais globalement bien réussi : si vous avez bloqué sur la partie qui vous demandait de redéfinir des fonctions précédentes en passant par fold, regardez la correction et entraînez vous à le refaire.
Dans la suite de l'année, lorsque l'on fera des fonctions sur les listes, vous pouvez prendre le réflexe de réfléchir à comment vous les implémenteriez en utilisant uniquement fold, map et filter (ce n'est pas toujours possible). De la même manière, même si l'on ne codera pas systématiquement en récursif terminal, c'est toujours un bon exercice de réfléchir à comment traduire les fonctions que l'on écrit vers du récursif terminal.

Concaténation Attention à ne pas utiliser la concaténation de manière abusive ! Par exemple, on ne va jamais concaténer une liste de taille 1 à une autre liste. En effet, si l'on veut concaténer à gauche :

```
1 let l = concatener [x] q
```

On écrira juste :

```
1 let l = x :: q
```

Et si l'on veut concaténer à droite, c'est un signe que l'on ne s'y prend pas de la bonne manière. Considérons la fonction suivante qui permet de renverser une liste :

```
1 let rec rev (l: 'a list) : 'a list =
2   match l with
3   | [] -> []
4   | x :: q -> concatener (rev q) [x]
```

Cette fonction est correcte, mais très coûteuse, car la concaténation est linéaire en la taille de la première liste, et donc la complexité C_n de la fonction `rev` vérifie $C_n = C_{n-1} + \Theta(n)$, i.e. $C_n = \Theta(n^2)$.

Exercice 4

Une astuce que j'ai vu dans plusieurs rendus qui simplifie beaucoup la fonction de comparaison : associer à chaque hauteur de carte un rang entier ($2 = 2$, ..., $10 = 10$, Valet $= 11$, Dame $=$ 12, Roi $= 13$, As $= 14$). On peut alors comparer les cartes avec l'ordre lexicographique (couleur, hauteur), et obtenir immédiatement l'ordre demandé (il faut avoir au préalable changé l'ordre de définition du type couleur pour correspondre à celui dans le sujet) :

```
1 let rang_carte (c: carte) : int = ...
2
3 let compare_carte (c1: carte) (c2: carte) : int =
4   let coul1, coul2 = couleur_de_carte c1, couleur_de_carte c2 in
5   let rang1, rang 2 = rang_carte c1, rang_carte c2 in
6   if c1 = c2 then 0
7   else if (coul1, rang1) < (coul2, rang2) then -1
8   else 1
```

Exercice 5

Dans plusieurs rendus, les fruits étaient représentés par un type somme, ce qui laissait un nombre fixé de fruits, par exemple :

```
type fruit = Pomme | Raisin | Ramboutan
type boisson =
  | Eau
  | JusDeFruit of fruit
  | BreizhCola of bool
  | Cocktail of float * boisson * boisson
```

D'autres élèves ont choisis de représenter directement les fruits par des chaînes de caractères :

```
type boisson =
  | Eau
  | JusDeFruit of string
  | BreizhCola of bool
  | Cocktail of float boisson * boisson
```

Pour la fonction `shaker`, on ne peut pas a priori traiter la liste vide, le cas de case sera une liste à un élément :

```
let rec shaker (l: boisson list): boisson =
  match l with
  | []    -> failwith "Shaker vide"
  | [x]   -> x
  | x::q  -> Cocktail (0.5, x, shaker q)
```