TP11: Listes, récursivité et types somme

MP2I Lycée Pierre de Fermat

Ce TP est à rendre avant le mercredi 14 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 :

— La valeur renvoyée en fonction des paramètres

— Les éventuelles hypothèses sur les 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 x mod y = 0
```

De plus, même si OCaml détecte automatiquement les types des différents objets que l'on définit, il est standard de préciser le type des fonctions, pour plus de lisibilité. Pour cela, on précise le type de chaque paramètre ainsi que le type de retour, en utilisant la syntaxe suivante :

```
let divise (x: int) (y: int) : bool = ...
```

On lit alors que x et y sont des entiers, et que la fonction renvoie un booléen.

On fera attention à **toujours** préciser ainsi le type des fonctions que l'on écrit, y compris si elles sont polymorphes. Par exemple, la fonction suivante renvoie la taille d'une liste :

```
(* Nombre d'éléments de l *)
let rec taille (l: 'a list) : int =
  match l with
  | [] -> 0
  | _ :: q -> 1 + taille q
```

Enfin, pour chaque fonction, on donnera un jeu de test. Pour cela, on peut utiliser `assert: bool -> unit`, qui fonctionne comme en C : elle prend en entrée un booléen et arrête le programme si ce booléen n'est pas `true`. Par exemple pour tester la fonction taille :

```
assert (taille [] = 0);;
assert (taille [2; 4; 5; 1; 1] = 5);;
```

De plus, le type de retour d'`assert` est `unit`, on peut donc enchaîner les assertions comme des print. Dans le fichier de chaque exercice, vous devez avoir une fonction `test: unit -> unit` qui teste toutes les fonctions via des assertions. Par exemple :

```
let test () =
  assert (divise 2 4);
  assert (not (divise 2 3));
  assert (divise 7 0);
  assert (taille [] = 0);
  assert (taille [2; 4; 5; 1; 1] = 5);
  print_string "Tous les tests ont réussi\n"
```

On peut alors lancer les tests dans utop après avoir importé le fichier en tapant simplement `test ();;`

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 (voir plus haut).

Exercice 1.

Étudions quelques fonctions sur les listes. Tentez de réutiliser au maximum les fonctions que vous définissez.

Q1. Écrivez une fonction `somme: int list -> int` renvoyant la somme d'une liste d'entiers.

Q2. Écrivez une fonction `recherche: 'a list -> 'a -> bool` qui, étant donné L une liste et x un élément, détermine si $x \in L$.

Q3. 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 $L = [x_1; \ldots; x_n; y_1; \ldots; y_m]$. Écrivez une fonction `concatener: 'a list -> 'a list -> 'a list` qui prend en entrée deux listes et renvoie leur concaténation. (*Indication : on pourra raisonner par récurrence sur la taille de* L_1*)*

Q4. Écrivez une fonction `multi_concat: 'a list list -> 'a list` prenant en entrée une liste de listes et renvoyant leur concaténation. Par exemple, `multiconcat [[1; 2]; [3]; [4; 5; 6]]` doit renvoyer `[1; 2; 3; 4; 5; 6]`.

Q5. 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. Par exemple, `map (fun x -> x*x)[2; 3; 4] = [4; 9; 16]`.

On s'intéresse maintenant à l'écriture d'une fonction qui **renverse** l'ordre d'une liste. On souhaite donc obtenir une fonction `reverse: 'a list -> 'a list`. On remarque qu'une liste OCaml se comporte comme une **pile**. On peut donc essayer d'écrire `reverse` en renverser la liste dans une autre. Nous allons donc utiliser une fonction auxiliaire, dont la spécification est :

```
(* Renverse l1, y concatène l2, et renvoie le résultat.
   Exemple: rev_concat [1; 2; 3] [4; 5; 6] = [3; 2; 1; 4; 5; 6]*)
let rec rev_concat (l1: 'a list) (l2: 'a list) : 'a list = ...
```

(Remarquons que dans l'exemple donné en commentaire, c'est comme si on avait une première pile contenant $1, 2, 3$, une deuxième contenant $4, 5, 6$, et que l'on avait versé la première dans la deuxième : on obtiendrait bien une pile contenant $3, 2, 1, 4, 5, 6$.)

Si l'on arrive à écrire cette fonction, alors on peut définir `reverse` par `let reverse l = rev_concat l []`.

Q6. Implémentez la fonction `rev_concat`, en raisonnant comme suit : on dépile un élément d'une pile et on l'empile dans l'autre.

Il est courant en OCaml de devoir coder une fonction en passant par une fonction auxiliaire, qui prend un paramètre en plus faisant office de "mémoire". Dans la fonction précédente, `rev_concat l1 l2` utilise l2 comme mémoire pour empiler successivement les éléments de l1.

Q7. En utilisant une fonction auxiliaire, écrivez une fonction `range: int -> int list` prenant en entrée un entier n et renvoyant la liste $[0; 1; \ldots; n-1]$.

Q8. En utilisant uniquement des fonctions que vous avez défini plus haut, écrivez une fonction `make_list: (int -> 'a)-> int -> 'a list` prenant en entrée une fonction f, un entier n, et renvoyant la liste $[f(0); f(1); \ldots; f(n-1)]$.

On s'intéresse à l'écriture d'une fonction `compose_liste` telle que `compose_liste [f1; f2; f3; ...; fn]` est la fonction $f_1 \circ f_2 \ldots f_n$. Si la liste est vide, alors le résultat sera la fonction identité.

Q9. Supposons que l'on a bien codé la fonction `compose_liste`. Quelles parties du code suivant ne sont pas bien typées en OCaml ? Pourquoi ? Que peut-on en déduire sur le type que doit avoir `compose_liste` ?

```
(* chaîne de caractère décrivant la parité de x *)
let pair (x: int) : string =
  if x mod 2 = 0 then "pair"
  else "impair"

(* chaîne doublant s. Exemple: repeter "bla" vaut "blabla" *)
let repeter (s: string) : string = s ^ s

let x = compose_liste [repeter; repeter] "salut"
let y = compose_liste [repeter; repeter; pair] 5
```

Q10. Écrivez la fonction `compose_liste`.

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 $F : n \mapsto F_n(0) + \cdots + F_n(n-1)$

Q11. En réutilisant le plus possible les fonctions définies dans le reste de l'exercice, fabriquer une fonction OCaml calculant les f_k, puis une fonction calculant F. Vérifiez $F(3) = 83, F(20) = 60321545051, F(30) = 12254504478085045$

Exercice 2.

Q1. Recopier le code des fonctions `taille`, `recherche`, `somme` et `multi_concat`.

Q2. Écrire une fonction `string_cat: string list -> string` qui renvoie la concaténation de toutes les chaînes de caractères de la liste donnée. On rappelle que l'opérateur `^` permet de concaténer deux strings.

On constate que toutes les fonctions des deux questions précédentes ont des définitions très proches, et on propose d'écrire une fonction qui permettrait de les généraliser. Cette fonction, très classique en programmation fonctionnelle, s'appelle ***fold*** (ou parfois ***reduce***) :

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

Q3. Recopiez la fonction fold, et évaluez les expressions suivantes :

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

fold (+) [1;2;3;4] 0;;
fold (fun x l ->x::l) [1;2;3;4] [];;
```

`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. Plus précisément, `fold f [x1;x2;...;xn] a` renvoie $f(x1, f(x2, \dots f(xn, a)\dots))$.

Par exemple, pour `fold (+) [1;2;3;4] 0`, le résultat est $1 + (2 + (3 + (4 + 0))) = 10$.

Q4. En utilisant fold, donnez une nouvelle définition des fonctions `somme`, `recherche` et `multi_concat`. A chaque fois, réfléchissez à :

- Quel est le résultat sur une liste vide : cela vous donne a
- Pour une liste `x::q`, si j'ai déjà le résultat sur q, comment est-ce que je mets à jour ce résultat avec x : cela vous donne f.

Q5. Donnez également une nouvelle définition de la fonction `compose_liste`.

Deux autres fonctions très courantes 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. Par exemple :

```
let est_pair (x: int) : bool =
  x mod 2 = 0

let l1 = filter est_pair [1;2;3;4;2;3;4] (* vaudra [2;4;2;4] *)
```

Q6. Définissez `filter` et `map`, d'abord directement, puis en utilisant `fold`

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.

Q7. Écrire une fonction `list_of_string: string -> char list` permettant de décomposer un string en liste de caractères.

On admet que la fonction suivante permet de transformer une liste de caractères en string :

```
let string_of_list (l: char list) : string =
  String.of_seq (List.to_seq l)
```

Q8. Écrire une fonction `split: string -> char -> string 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, et qui stocke dans `curr` les caractères lus depuis la dernière occurrence du séparateur. On signale également l'existence de la fonction `List.rev` qui permet de renverser l'ordre d'une liste.

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 c1 = Pique ;;
let t = (Carreau, 2, "bla");;
let est_rouge (c: couleur) : bool = match c with
  | Coeur | Carreau -> true (* Coeur ou bien Carreau *)
  | Pique | Trefle -> false

assert (est_rouge Carreau)
```

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

```
type carte =
  | Nombre of (int * couleur) (* Nombre (2, Coeur) est le 2 de coeur, et ainsi de suite *)
  | Valet of couleur (* Valet Pique est le valet de pique *)
  | 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: carte) : couleur = match ca with
  | Nombre (_, c) | Valet c
  | Dame c | Roi c | As c-> c ;;

assert (couleur_de_carte (Roi Trefle) = Trefle) ;;
```

Les mots `Nombre`, ... `As` sont appelés des ***constructeurs***. Ce ne sont pas des fonctions mais ils s'utilisent de manière assez semblable ! 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 vues jusqu'à maintenant.

Q1. Écrivez une fonction `string_of_couleur: couleur -> string` qui renvoie le nom d'une couleur sous forme de chaîne de caractère

Q2. 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`.

Q3. Écrire une fonction

Q4. Tentez de comparer quelques cartes avec $<$, $<=$, $=$, etc... que remarquez vous sur l'ordre natif d'OCaml ?

Q5. Écrivez une fonction `cmp: carte -> carte -> int` qui permet de comparer deux cartes selon l'ordre suivant (plus compatible avec les jeux de cartes) : tous les coeurs sont plus petits que tous les carreaux, qui sont plus petits que tous les piques, qui sont plus petits que tous les trèfles. Au sein d'une couleur, l'ordre est 2, 3, ..., 10, valet, dame, roi, as. La fonction renverra $-1, 0$ ou 1 selon si la première carte est inférieure, égale ou supérieure à la première.

Q6. Écrivez une fonction `insert: carte -> carte list -> carte list` qui permet d'insérer une carte au bon endroit dans une main supposée triée.

Q7. Utilisez cette fonction pour écrire une fonction `insert_sort: carte list -> carte list` implémentant le tri par insertion et permettant de trier une main de cartes.

Q8. 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).

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.

Q1. Créer un type `type boisson = ... ;;` permettant de représenter les boissons.

Q2. 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'empêche un type d'être récursif, c'est à dire d'avoir un constructeur utilisant le type lui-même. On peut par exemple rajouter au type boisson le cas suivant :

```
1 type boisson =
2   ...
3   | Cocktail of boisson * float * boisson * float
```

On souhaite donner à `Cocktail (b1, b2, p)` le sens "boisson contenant une proportion $p \in [0, 1]$ de boisson b_1 et le reste de boisson b_2".

Q3. Modifier la boisson de calcul de prix pour prendre en compte ce nouveau constructeur.

Q4. 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
```

Dans la suite, on appelle **boisson de base** toute boisson n'étant pas un cocktail.

Q5. Écrire une fonction `string_of_boisson` calculant le nom d'une boisson de base.

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

Q6. 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 :

```
ingredients (Cocktail(Eau, Cocktail (Breizh, Jus "pomme", 0.4), 0.5));;
(* Vaut [(Eau, 0.5); (Breizh, 0.2); (Jus pomme, 0.3)] *)

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

Indication : on pourra utiliser une fonction auxiliaire `ingredients_fraction` de signature `boisson -> float -> (boisson*float)list` prenant en argument additionnel une proportion p indiquant la proportion totale de la boisson. Par exemple :

```
ingredients_fraction (Cocktail(Eau, Cocktail (Breizh true, Jus "pomme", 0.4), 0.5)) 0
    .7;;
(* Vaut [(Eau, 0.35); (Breizh, 0.14); (Jus pomme, 0.21)], chaque proportion a été
    multipliée par 0.7 *)
```

Le module `List` d'OCaml possède de nombreuses fonctions utiles, notamment une fonction `List.sort: ('a -> 'a -> int)-> 'a list -> 'a list` telle que `List.sort cmp l` trie la liste `l` selon la fonction de comparaison `cmp` (qui doit renvoyer un entier négatif, nul ou positif selon le résultat de la comparaison). De plus, OCaml possède une fonction de comparaison native, `compare: 'a -> 'a -> int`. Comme nous l'avons vu sur l'exercice précédent, l'ordre natif d'OCaml sur les types somme est de comparer d'abord les constructeurs, puis ensuite les arguments. Ainsi, si on a une liste de boissons `l: boisson list`, alors `List.sort compare l` est une copie de `l` où les boissons sont triées, et donc regroupées par constructeur.

Q7. Écrire une fonction `agreg_sum: ('a * float)list -> ('a * float)list` qui prend en entrée une liste de couples, et qui regroupe les couples selon la première composante, en sommant les deuxième composantes. On supposera que la liste d'entrée est triée. Par exemple :

```
agreg_sum [("bla", 0.1); ("bla", 0.3); ("truc", 0.4);("truc",0.2)] ;;
(* Vaut [("bla", 0.4); ("truc", 0.6)] *)
```

Indication : on pourra utiliser une fonction auxiliaire ayant deux paramètres supplémentaires donnant l'élément actuel et la somme actuelle pour cet élément.

Q8. En vous aidant des fonctions précédentes, écrire une fonction `recette: boisson -> unit` qui affiche la recette d'une boisson selon le format décrit plus haut.

Exercice 5.

Plus tôt dans le TP, nous avons implémenté un algorithme de tri par insertion. On étudie maintenant deux algorithmes plus performants : le tri rapide et le tri fusion.

Q1. Écrire une fonction `est_triee: 'a list -> bool` déterminant si son entrée est triée dans l'ordre croissant.

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

Algorithme 1 : TriRapide

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
$p, Q \leftarrow$ tête de L, queue de L ;
$L_{\leq} \leftarrow$ Liste des éléments $y \in Q$ tels que $y \leq p$;
$L_{>} \leftarrow$ Liste des éléments $y \in Q$ tels que $y > p$;
$L'_{\leq} \leftarrow$ **TriRapide**$(L_{\leq})$;
$L'_{>} \leftarrow$ **TriRapide**$(L_{>})$;
$L' \leftarrow$ la concaténation de $L'_{\leq}, [p]$ et $L'_{>}$;
retourner *L'*

Q2. Écrivez une fonction `partition: '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.

Q3. Écrivez une fonction `tri_rapide: 'a list -> 'a list` triant sa liste d'entrée. Donnez également un jeu de test pour vérifier votre fonction.

Passons au tri fusion. Ce tri repose sur l'utilisation de la fonction suivante :

```
1 (* Renvoie une liste triée contenant les éléments de l1 et l2.
2    Préconditions: l1 et l2 sont triées *)
3 let rec fusion (l1: 'a list) (l2: 'a list) = ...
```

Q4. Implémentez la fonction `fusion`, puis utilisez la pour implémentez une fonction `tri_fusion: 'a list -> 'a list` qui trie son entrée en la divisant en deux listes égales, en les triant récursivement puis en fusionnant les listes triées. *Indication : il existe différentes manières de séparer une liste en deux. Si vous n'avez pas d'inspiration, imaginez que vous distribuez des cartes.*

Exercice 6.

Les langages fonctionnels se prêtent bien à l'écriture de compilateurs et d'interpréteurs, car il est facile de représenter les arbres syntaxiques. On propose pour commencer le type suivant pour représenter des expressions simples :

```
type expr =
  | Const of float (* constante *)
  | Add  of expr * expr (* Add(e1, e2) correspond àe1 + e2 *)

(*** Exemples: ***)
(* représentation de 3.2 + 4 *)
let e1 = Add(Const 3.2, Const 4.)

(* représentation de (1 + 2) + (3 + (4 + 5)) *)
let e2 =
Add(
  Add(
    Const 1.,
    Const 2.
  ),
  Add(
    Const 3.,
    Add(
      Const 4.,
      Const 5.
    )
  )
)
```

Q1. Écrivez une fonction `eval: expr -> float` qui évalue une expression.

Q2. Ajoutez un constructeur `Mul` au type `expr`, servant à représenter le produit de deux expressions, et modifiez la fonction `eval` en conséquence

Nous allons rajouter à nos expressions la possibilité d'avoir des variables. Les listes de type `(string * int)list` serviront à représenter un ***contexte***, c'est à dire une association entre les variables et les entiers :

```
type context = (string * int) list

(* x -> 2.0, r -> -0.2 *)
let c1 = [("x", 2.0); ("r", -0.2)]
```

Q3. Écrire une fonction `get_var: (string -> context -> int` telle que `get_var s l` cherche dans l un couple (s, n) et renvoie l'entier n correspondant.

La fonction précédente existe déjà dans OCaml : `List.assoc: 'a -> ('a * 'b)list -> 'b` prend en entrée un élément x et une liste L et renvoie le premier y tel que (x, y) apparaît dans x (et lève une erreur s'il n'y en a pas).

Q4. Modifier le type des expressions pour y rajouter le cas `| Var of string` représentant les variables. Modifier la fonction `eval` pour qu'elle prenne également un contexte en paramètre, et pour qu'elle traite le nouveau cas ajouté au type.

On souhaite maintenant agrémenter nos expressions avec une construction if-then-else. Pour cela, nous allons créer un deuxième type, pour les expressions booléennes :

```
type boolexpr =
  | BConst of bool (* constantes true et false *)
  | Or of boolexpr * boolexpr (* OU booléen *)
  | And of boolexpr * boolexpr (* ET booléen *)
  | Not of boolexpr (* NON booléen *)
  | Eq of expr * expr (* égalité *)
  | Leq of expr * expr (* inférieur ou égal *)
```

On rajoute également le constructeur suivant au type expr :

```
| IFTE of boolexpr * expr * expr (* if b then e1 else e2 *)
```

Comme les deux types dépendent l'un de l'autre, on doit les définir en utilisant le mot clé `and` :

```
type expr =
  ...

and boolexpr =
  ...
```

De la même manière, nous allons définir des fonctions sur ces deux types qui iront par paires et dépendront l'une de l'autre, il faudra alors aussi utiliser le mot clé **and**. Par exemple, les fonctions suivantes comptent le nombre d'occurrences d'une variable dans une expression / dans une expression booléenne :

```
let rec var_count (e: expr) (v: string) : int =
  match e with
  | Const _ -> 0
  | Var x -> if x = v then 1 else 0
  | Add (e1, e2) | Mul (e1, e2) -> var_count e1 v + var_count e2 v
  | IFTE (b, e1, e2) -> var_count_bool b v + var_count e1 v + var_count e2 v

and var_count_bool (b: boolexpr) (v: string) : int =
  match b with
  | BConst _ -> 0
  | Not bb -> var_count_vool bb v
  | And (b1, b2) | Or (b1, b2) -> var_count_bool b1 v + var_count b2 v
  | Eq (e1, e2) | Leq (e1, e2) -> var_count e1 v + var_count e2 v
```

Q5. Modifier `eval` pour qu'elle soit définie en même temps qu'une fonction `eval_bool` équivalente.