

# TP10: OCaml: récursivité et types

## sommes

Lorsque vous écrivez une fonction OCaml, vous devez la **typer**, et écrire un commentaire qui précise la valeur renvoyée par la fonction par rapport aux paramètres, ainsi que les préconditions / hypothèses nécessaires. Par exemple :

```
1 (* true si x divise y, false sinon.  
2    x et y doivent être positifs, et y doit être non-nul. *)  
3 let divise (x: int) (y: int) : bool =  
4   if x <= 0 then failwith "Argument invalide"  
5   else y mod x = 0  
6  
7 (* Nombre d'éléments de l *)  
8 let rec taille (l: 'a list) : int =  
9   match l with  
10  | [] -> 0  
11  | _ :: q -> 1 + taille q
```

Enfin, pour chaque fonction, on donnera un jeu de test. Pour cela, on peut utiliser la fonction `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`, on créera une fonction `test_taille: unit -> unit` :

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

On écrira aussi une fonction `test: unit -> unit` permettant de lancer tous les tests :

```
1 let test () =  
2   test_divise ();  
3   test_taille ();  
4   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 ();;`

**ATTENTION** : Il ne faut pas mettre de point-virgule à la fin de la dernière instruction dans une fonction : le point virgule se met **ENTRE** les instructions!!

## Exercice 1: 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 la fonction `taille` plus haut).

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

- 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  $L_1 = [x_1; \dots; x_n]$  et  $L_2 = [y_1; \dots; y_m]$ , la concaténation de  $L_1$  et  $L_2$  est  $L = [x_1; \dots; x_n; y_1; \dots; y_m]$ . Écrivez une fonction `concatener` qui prend en entrée deux listes  $L_1$ , et  $L_2$  et renvoie leur concaténation. (*Indication : on pourra raisonner par récurrence sur  $L_1$* )
- Q4.** Écrivez une fonction `multi_concat: 'a list list -> 'a list` prenant en entrée une liste de listes et renvoyant leur concaténation.  
Par exemple, `multi_concat [[1; 2]; [3]; [4; 5; 6]] = [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 `rev: 'a list -> 'a list`. Nous avons déjà fait la remarque qu'une liste OCaml se comporte comme une **pile**. Nous allons donc essayer d'écrire `rev` en renversant la liste d'entrée dans une autre. Pour cela, nous allons passer par une fonction auxiliaire, dont la spécification est :

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

- Q6.** Implémentez la fonction `rev_concat` par récurrence sur `l1`.
- Q7.** En déduire la fonction `rev: 'a list -> 'a list`.

La fonction `rev` nous a demandé de coder une fonction auxiliaire utilisant un paramètre additionnel permettant de “stocker le résultat”. On dit que `l2` est un **accumulateur**.

- Q8.** En utilisant une fonction auxiliaire et un accumulateur, écrivez une fonction `range: int -> int list` prenant en entrée  $n \in \mathbb{N}$  et renvoyant la liste  $[0; \dots; n-1]$ .

## Exercice 2: Fonctions classiques sur les listes

**Q1.** Copiez-collez le code des fonctions `taille`, `recherche`, `somme` et `multi_concat`.

**Q2.** Écrivez 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'étudier une fonction qui permet de les généraliser. Cette fonction, classique en programmation fonctionnelle, s'appelle **fold** (ou parfois **reduce**) :

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

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

```
1 fold (fun x y -> x^y) ["vive"; " "; "OCaml"; "!!!"] "";
2 (* les opérateurs sont des fonctions, on peut donc également écrire: *)
3 fold (^) ["vive"; " "; "OCaml"; "!!!"] "";
4
5 fold (+) [1;2;3;4] 0;;
6 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  $b$  et d'une fonction d'agrégation  $f$ . Plus précisément, `fold f [x1;x2;...;xn] a` renvoie  $f(x_1, f(x_2, \dots f(x_n, b) \dots))$ .

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  $b$
- 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  $f$ .

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 :

```
1 let est_pair (x: int) : bool =
2   x mod 2 = 0
3
4 let l1 = filter est_pair [1;2;3;4;2;3;4] (* vaudra [2;4;2;4] *)
```

**Q5.** Définissez `filter` et `map`, d'abord directement, puis en utilisant `fold`

**Q6.** En utilisant `filter`, `map`, `fold` et la fonction `range` de l'exercice précédent, écrire une fonction `sum_div: int -> int` non-récursive calculant la somme des carrés des diviseurs d'un entier non nul.

### Exercice 3: Chaînes de caractères

Intéressons nous à la manipulation des strings. Pour accéder au  $k$ -ème caractère d'une string  $s$  de longueur  $n$ , on utilise la syntaxe `s.[k]` (avec  $0 \leq k < n$ ). 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'une string.

**Q1.** Écrire une fonction `list_of_string: string -> char list` permettant de décomposer une string en liste de caractères. On pourra passer par une fonction auxiliaire prenant aussi en entrée un indice permettant d'itérer sur la string.

**Documentation** Sur la machine virtuelle, le logiciel **Zeal** contient la documentation d'OCaml et de sa librairie standard. Vous y trouverez donc les descriptions des fonctions du module `String` et de tous les autres modules accessibles par défaut. Si vous n'êtes pas sur la machine virtuelle, vous pouvez trouver la documentation en ligne : [v2.ocaml.org/api](http://v2.ocaml.org/api).

Notons que la librairie standard contient un module `List`, dans lequel se trouvent de nombreuses fonctions utiles sur les listes, dont certaines que l'on a codé dans les exercices précédents :

- `List.length: 'a list -> int` pour la longueur d'une liste ;
- `List.map: ('a -> 'b) -> 'a list -> 'b list` pour appliquer une fonction à une liste, élément par élément ;
- `List.filter: ('a -> bool) -> 'a list -> 'a list` pour filtrer les éléments d'une liste vérifiant un prédicat donné ;
- etc...

**Q2.** Cherchez la documentation de la fonction `String.split_on_char` et testez-la.

Nous allons réimplanter cette fonction par nos propres moyens. On admet que la fonction suivante permet de transformer une liste de caractères en string :

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

**Q3.** Écrire une fonction `split: string -> char -> string list` permettant de diviser une string en mots, selon un caractère de séparation. Par exemple :

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

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

```
1 let rec split_from_i (s:string) (sep:char) (i:int) (curr:char list) : string
2 list=
3   ...
```

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.

## Exercice 4: Cartes à jouer

Nous allons utiliser des types sommes pour représenter et manipuler des cartes à jouer. On rappelle qu'un type somme permet de représenter un ensemble contenant plusieurs sous-catégories. Par exemple :

```
1 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 :

```
1 let c1 = Pique
2 let t = (Carreau, 2, "bla")
3 let est_rouge (c: couleur) : bool = match c with
4   | Coeur | Carreau -> true (* Coeur ou bien Carreau *)
5   | Pique | Trefle -> false
6 ;;
7 assert (est_rouge Carreau) ;;
```

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

```
1 type tete = Valet | Dame | Roi
2
3 type carte =
4   | Nombre of (int * couleur) (* Nombre (2, Coeur) est le 2 de coeur, et ainsi de suite *)
5   | Tete of (tete*couleur) (* Tete (Valet, Pique) est le valet de pique *)
6   | Joker
7
8 let carte_1 = Tete(Valet, Coeur)
9 let carte_2 = Nombre (9, Pique)
10
11 (* Renvoie la couleur d'une carte *)
12 let couleur_de_carte (ca: carte) : couleur = match ca with
13   | Nombre (_, c) -> c
14   | Tete (_, c) -> c
15   | Joker -> failwith "pas de couleur"
16 ;;
17 assert (couleur_de_carte (Roi Trefle) = Trefle) ;;
```

Les mots `Nombre`, ... `Roi` sont appelés des *constructeurs*. Un nom de constructeur doit commencer par une majuscule, et un nom de type doit commencer par une minuscule.

Attention, **un constructeur n'est pas une fonction**, on ne pourrait pas écrire le code suivant pour créer un 5 de pique :

```
1 let a = Nombre (* Erreur de syntaxe: Nombre attend un paramètre *)
2 let b = a (5, Pique)
```

**Q1.** Recopiez les types `couleur`, `tete`, `carte`.

**Q2.** Écrivez une fonction `string_of_couleur: couleur -> string` qui renvoie le nom d'une couleur sous forme de chaîne de caractère

**Q3.** 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...  
**Le 1 doit s'appeler "As" !**

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

**Q4.** Tentez de comparer quelques cartes avec `<`, `<=`, `=`, etc... et proposez une description précise de l'ordre par défaut d'OCaml.

**Q5.** Écrivez une fonction `compare_carte: carte -> carte -> int` qui permet de comparer deux cartes selon un ordre plus naturel :

- Les couleurs sont rangées ensembles : 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 deuxième.

**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 l'ordre que vous voulez).

## Exercice 5: La soif

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 * 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  $1 - p$  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  $B_1 \dots B_n$  et faisant un gigantesque cocktail, de la forme :

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

On voudrait pouvoir afficher la recette d'un cocktail, 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**.

- Q6.** Écrire une fonction `ingrédients: 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 ingrédients (Cocktail(Eau, Cocktail (Breizh, Jus "pomme", 0.4), 0.5));;
2 (* Vaut [(Eau, 0.5); (Breizh, 0.2); (Jus pomme, 0.3)] *)
3
4 ingrédients (Cocktail(Eau, Eau, 0.5));;
5 (* Vaut [(Eau, 0.5); (Eau, 0.5)] *)
```

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

```
1 ingrédients_fraction (Cocktail(Eau, Cocktail (Breizh true, Jus "pomme", 0.4),
2   0.5)) 0.7;;
3 (* Vaut [(Eau, 0.35); (Breizh, 0.14); (Jus pomme, 0.21)], chaque proportion a
4   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 :

```
1 agreg_sum [("bla", 0.1); ("bla", 0.3); ("truc", 0.4); ("truc", 0.2)] ;;
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 6: Tri rapide et tri fusion

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  $L$

- 1 **si**  $L$  *est de taille 0 ou 1* **alors**
- 2     **retourner**  $L$
- 3  $p, Q \leftarrow$  tête de  $L$ , queue de  $L$  ;
- 4  $L_{\leq} \leftarrow$  Liste des éléments  $y \in Q$  tels que  $y \leq p$  ;
- 5  $L_{>} \leftarrow$  Liste des éléments  $y \in Q$  tels que  $y > p$  ;
- 6  $L'_{\leq} \leftarrow \text{TriRapide}(L_{\leq})$ ;
- 7  $L'_{>} \leftarrow \text{TriRapide}(L_{>})$ ;
- 8  $L' \leftarrow$  la concaténation de  $L'_{\leq}$ ,  $[p]$  et  $L'_{>}$ ;
- 9 **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 de même longueur, 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 7

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

```

1 type expr =
2   | Const of float (* constante *)
3   | Add of expr * expr (* Add(e1, e2) correspond à e1 + e2 *)
4
5 (** Exemples: **)
6 (* représentation de 3.2 + 4 *)
7 let e1 = Add(Const 3.2, Const 4.)
8
9 (* représentation de (1 + 2) + (3 + (4 + 5)) *)
10 let e2 =
11   Add(
12     Add(
13       Const 1.,
14       Const 2.
15     ),
16     Add(
17       Const 3.,
18       Add(
19         Const 4.,
20         Const 5.
21       )
22     )
23   )

```

**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 * float) list` serviront à représenter un *contexte*, c'est à dire une association entre les variables et les entiers :

```

1 type context = (string * float) list
2
3 (* x -> 2.0, r -> -0.2 *)
4 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 augmenter nos expressions avec une construction if-then-else. Pour cela, nous allons créer un deuxième type, pour les expressions booléennes :

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

On rajoute également le constructeur suivant au type expr :

```
1 | 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` :

```
1 type expr =
2   ...
3
4 and boolexpr =
5   ...
```

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 :

```
1 let rec var_count (e: expr) (v: string) : int =
2   match e with
3   | Const _ -> 0
4   | Var x -> if x = v then 1 else 0
5   | Add (e1, e2) | Mul (e1, e2) -> var_count e1 v + var_count e2 v
6   | IFTE (b, e1, e2) -> var_count_bool b v + var_count e1 v + var_count e2 v
7
8 and var_count_bool (b: boolexpr) (v: string) : int =
9   match b with
10  | BConst _ -> 0
11  | Not bb -> var_count_bool bb v
12  | And (b1, b2) | Or (b1, b2) -> var_count_bool b1 v + var_count b2 v
13  | 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.
- Q6.** Ajouter la possibilité de faire des `let in` (non récursifs) dans les expressions, en ajoutant un constructeur `Let of string * expr * expr` tel que `Let (v, e1, e2)` représente l'expression `let v = e1 in e2`.
- Q7.** Plutôt que d'avoir deux types distincts pour les expressions arithmétiques et booléennes, proposer un type unique `expr` fusionnant les deux. Expliquer les difficultés qui surviennent, et proposer d'éventuelles solutions.