université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

jeudi 7 janvier 2021

Introduction à la programmation fonctionnelle
L2 Informatique

Examen

Durée 2h00, tiers temps additionnel 40 minutes

Consignes l'examen dure 2h00 et est sur 20 points. Aucun document n'est autorisé. Un aide mémoire OCaml est disponible à la page 8. Le barème est indicatif et proportionnel à la difficulté des exercices.

Conditions spéciales d'examen en raison de la situation sanitaire et des règles en vigueur :

- le port du masque en salle d'examen comme sur le reste du campus est obligatoire
- ne pas anonymiser vos copies
- il est interdit d'échanger du matériel entre étudiants
- signer une fois la feuille d'émargement avec votre stylo personnel, lors du passage du surveillant
- déposez votre carte d'étudiant ou pièce d'identité sur la table, de façon à ce qu'elle puisse être lue par le surveillant
- votre copie est ramassée à votre place par les surveillants.
- une fois la copie rendue, merci de quitter la salle sans provoquer d'attroupement et dans le respect de la distanciation

Le non respect de ces règles peut entrainer l'exclusion **immédiate** de la salle d'examen.

1 Lecture de code, typage (4 points)

Pour chacun des fragments de codes ci-dessous, donner :

- soit le type de toutes les variables globales
- soit uniquement une erreur de type à l'endroit où elle se produit.

On ne demande pas de justification dans le premier cas. Les deux exemples ci-dessous illustrent le type de réponse attendu :

Exemple 1

```
1 let f y = y + 1;;
2 let x = f (f 42);;
```

Exemple 2

```
1 let f x = x + 1;;
2 let g y = "42" - (f y) ;;
```

Pour l'exemple 1 on attend la réponse :

- `f : int -> int`
- `x : int`

Pour l'exemple 2 on attend la réponse :

- erreur de typage ligne 2 : la soustraction attend 2 entiers mais est appliquée à une chaîne et un entier.

Questions

(1)

```
1 let f y = y *. 3.14;;
2 let y = f (float 42);;
```

(2)

```
1 let f y = 1 + y;;
2 List.map f [ 2.0; 3.0 ];;
```

(3)

```
1  let f g l =
2    let rec loop l acc1 acc2 =
3       match l with
4       [] -> (acc1, acc2)
5       | p :: ll ->
6         if g p then
7           loop ll (1+acc1) acc2
8         else
9           loop ll acc1 (1+acc2)
10   in
11   loop l 0 0 ;;
```

Réponse:

1. — `f : float -> float`
   — `y : float`
2. Erreu de typage, la fonction `List.map` attend une fonction prenant en argument des valeurs du même type que les éléments de la liste. La fonction `f` attend des entiers, la liste contient des `float`.
3. On n'attendait pas cette explication, mais voici comment chercher cette question au brouillon.

```
f : X -> Y -> Z  (car f attend 2 arguments)
g : X (le type du premier argument)
l : Y (le type du deuxième argument)

On a une fonction interne loop qui prend trois arguments :
loop : A -> B -> C -> D
l (de loop) : A
acc1 : B
acc2 : C

On remarque que loop fait un match sur l pour savoir si c'est une
liste vide ou non, donc :
A = E list

On remarque que l'on fait 1+acc1 et 1+acc2 donc :
B = int
C = int
C'est confirmé par la ligne 11 qui appelle loop avec 0 et 0
comme valeurs initiales de acc1 et acc2.
On voit dans le cas de base de loop qu'elle renvoie (acc1, acc2) donc
D = (int * int) ( la fonction loop renvoie une paire d'entiers).

On voit que g est appelé sur chaque élément de la liste et
                qu'elle
renvoie un booléen (utilisé dans un if ligne 6).
Donc :
X = E -> bool

Enfin, on voit que le type de retour de f est le même que celui de

loop, on peut donc conclure :
— f : ('a -> bool) -> 'a list -> (int * int)
```

Remarque : cette fonction prend une fonction de test en argument, une liste de valeurs à tester et renvoie le couple d'entier contenant le nombre de valeurs pour lesquelles le test est vrai et le nombre pour lequel le test est faux.

2 Fonctions récursives (6 points)

Dans cet exercice, on écrira des fonctions récursives, sans utiliser les itérateurs du module `List`. Il est évidemment autorisé (et parfois nécessaire) d'utiliser des fonctions imbriquées auxiliaires.

1. (1 point) écrire une fonction récursive `log2 : int -> int` qui renvoie la partie entière du logarithme en base 2 de son argument (i.e. le nombre de fois où il faut le diviser par 2 pour obtenir 1). Si l'argument est nul ou négatif, la fonction lève une exception avec `failwith "erreur"`. Par exemple :
   — `log2 8` donne `3` (car $8/2/2/2 = 1$).
   — `log2 16` donne `4` (car $16/2/2/2/2 = 1$).
   — `log2 25` donne aussi `4` (car $25/2/2/2/2 = 1$).
   — `log2 0` lève une erreur.

Réponse:

```
let rec log2 n =
 if n <= 0 then failwith "erreur"
 else if n = 1 then 0
       else 1 + (log2 (n/2))
 ;;
```

2. (1.5 point) écrire une fonction récursive `somme : (int -> int) -> int -> int` telle que `somme f n` calcule la somme des `(f i)` pour `i` compris entre `0` et `n` inclus. Votre fonction doit être récurisve terminale.

Réponse: Cette fonction est issue du cours la seule difficulté ici est de qu'on veut la version récursive terminale :

```
let somme f n =
   let rec loop n acc =
       if n < 0 then acc
       else loop (n-1) (acc + (f n))
   in
    f n 0
 ;;
```

La fonction interne utilise un accumulateur et « accumule » la somme à chaque appel récursif. La moitié des points sont donnés si la fonction est correcte mais non récursive terminale.

3. (2 points) écrire une fonction récursive `sous_liste : int -> int -> 'a list -> 'a list` telle que `sous_liste i j l` renvoie une liste contenant les éléments de `l` se trouvant entre les positions `i` et `j`. On suppose que $0 \leq$ `i` $\leq$ `j` $<$ `List.length l` (vous n'avez pas à le vérifier). On pourra utiliser une fonction auxiliaire prenant en argument un entier `k` supplémentaire représentant la position courante dans la liste.

Réponse: On utilise l'indication données. La fonction interne parcours la liste, tant que k est inférieur à i, on s'appeller récursivement sur la suite pour sauter le début. Puis tant que k est inférieur à j on ajoute au résultat. Enfin dès qu'on dépasse k on peut s'arrêter. On note que cette fonction n'est pas récursive terminale, mais ça n'était pas demandé.

```
let sous_liste i j l =
  let rec loop k l =
  match l with
     [] -> failwith "erreur" (* la liste est trop courte, mais ça ne
                                devrait pas arriver *)
   | p :: ll -> if k < i then loop (k+1) ll
                else if k <= j then p :: (loop (k+1) ll)
                else []
```

4. (1.5 point) écrire une fonction récursive `zip : 'a list -> 'a list -> 'a list` prenant en argument deux listes supposées être de même longueur et qui les fusionne. Si `l1 = [ 1;2;3;4;5]` et `l2 = [10;11;12;13;14]` alors `zip l1 l2 = [ 1;10;2;11;3;12;4;13;5;14 ]`. C'est à dire que les éléments de `l1` et `l2` sont placés alternativement dans la liste résultante, en commençant par `l1`. Si on détecte qu'une liste n'est pas de la bonne longueur, la fonction doit lever une exception avec `failwith "erreur"`.

Réponse: Il suffit de parcourir les deux listes en même temps :

```
    let rec zip l1 l2 = match l1, l2 with
     | [], [] -> []
     | [], _ :: _ | _::_, [] -> (* l'une est vide et pas l'autre *)
                                failwith "erreur"
     | p1:: ll1, p2::ll2 -> p1::p2:: (zip ll1 ll2)
;;
```

3 Problème : validation de documents XML (10 points)

Attention, dans cet exercice, certaines questions réutilisent des fonctions écrites dans les questions précédentes. Vous pouvez utiliser ces fonctions même si vous n'avez pas réussi à les écrire.

Contexte

Le format XML est un format texte permettant de structurer des données. En voici un exemple :

```
<a>
  <b>Hello, ça va ?<c id='42'>oui ça va</c></b>
  <b>Tant<toto id='55' val='foo'>mieux!</toto></b>
</a>
```

Les documents HTML, utilisés pour les pages Web, sont des exemples de tels documents. On appelle un texte tel que `<a>` ou `<toto id='55' val='foo'>` une balise **ouvrante**. Un texte tel que `</b>` est une balise **fermante**. Un texte tel que `id='55'` est un **attribut**. Les autres caractères (par exemple « `Hello, ça va ?` ») sont du texte simple.

Les contraintes sont les suivantes :

— les balises ouvrantes et fermantes sont bien parenthésées, c'est à dire que des séquences comme

```
<a>
  <b> </Toto>
   <c>
</a>
```

sont interdites. En effet, la balise ouvrante `<b>` est refermée par une balise fermante différente (`</Toto>`) et la balise ouvrante `<c>` n'est pas fermée.

— les noms d'attributs doivent êtres uniques au sein d'une balise ouvrante, on n'a donc pas le droit d'avoir des balises comme :

```
<a id='52' toto='foo' id='54'></a>
```

car l'attribut `id` apparaît deux fois.

Représentation en OCaml

On se donne les types OCaml suivants :

```
type elem =
    BO of string * (string * string) list
  | BF of string
  | T of string
  ;;
type doc = elem list
;;
```

Le type **elem** représente les éléments d'un fichier XML : balise ouvrante avec nom et attributs, balise fermante ou texte simple. Le type **doc** représente un document comme une liste d'éléments. On suppose qu'il existe une fonction qui crée une valeur de type **doc** à partir d'un fichier texte. Par exemple, le document donné en introduction correspond à la valeur OCaml (l'indentation n'est là que pour aider la lecture) :

```
  [ BO ("a", []);
    BO ("b", []);  T ("Hello, ça va ?"); BO("c", [ ("id", "42") ]);
                   T ("oui, ça va");  BF("c"); BF("b");
    BO ("b", []);  T ("Tant"); BO ("toto", [ ("id", "55"); ("val", "foo")]);
                   T ("mieux!"); BF("toto"); BF("b");
    BF ("a")]
```

On suppose que les retours à la ligne et les blancs du fichier ne sont pas significatifs et sont donc ignorés.

Le but de l'exercice est d'écrire des fonctions utilitaires pour réafficher un document ou vérifier les contraintes de bonne formation.

Questions

Dans toutes les questions, il est vivement recommandé d'utiliser les fonctions prédéfinies sur les listes, sauf lorsque l'énoncé indique d'écrire explicitement une fonction récursive.

1. (1 point) Donner la valeur OCaml correspondant au document :

```
<xx>
  <yy bar='42'>Hello, world!</yy>
  <zz>Yes</zz>
</xx>
```

Réponse: On utilise ici l'indentation pour aider la lecture :

```
  [
    BO ("xx", []);
      BO ("yy", [ ("bar", 42)]); T("Hello, world!"); BF("yy")
      BO ("zz", []); T("Yes"); BF("zz");
   BF("xx")
 ]
```

2. (1 point) Écrire une fonction `pr_att_list : (string * string) list -> unit` qui affiche une liste de paires de chaînes de caractères dans la console comme des attributs. Pour chaque paire de chaîne il faut donc afficher : un espace, la première chaine, un symbole égal, un guillemet simple, la seconde chaîne et un guillemet simple. Par exemple :

```
pr_att_list [("a", "25"); ("id", "toto"); ("foo", "bar")]
```

affiche :

```
a='25' id='toto' foo='bar'
```

Réponse: À retenir : la plupart les fonctions qui affichent des listes de choses vont utiliser `List.iter` :

```
let pr_att_list l =
 List.iter (fun att -> Printf.printf "\%s='\%s'" (fst att) (snd att)) l
```

3. (1.5 point) Écrire une fonction `pr_elem : elem -> unit` qui affiche une valeur de type `elem` dans la console.
   — Le nom des balises ouvrantes est affiché entre < et >, avec éventuellement la liste d'attributs (utiliser la fonction prédécente).
   — Le nom des balises fermantes est affiché entre </ et >.
   — Les textes simples sont directement affichés sans guillemet.

Réponse: Ici on n'affiche pas une liste mais un type somme définit pour l'exercice (comme par exemple le type des cartes dans le cours). On utilise un **`match with`** :

```
    let pr_elem e =
    match e with
      T (s) -> Printf.printf "\%s" s
      | BO (n, att) -> Printf.printf "<\%s" n;
                       pr_Att_list att;
                       Printf.printf ">"
      | BF (n) -> Printf.printf "</\%s>" n
```

4. (1 point) Écrire une fonction `pr_doc : doc -> unit` qui affiche un document dans la console.

Réponse: On remarque que le type `doc` est un simple alias vers le type list. On veut donc afficher des listes d'éléments, on utilise encore une fois `List.iter` :

```
      let pr_doc l =  List.iter pr_elem l
```

5. (2.5 point) Écrire une fonction `verif_att_list : (string * string) list -> bool` qui vérifie qu'une liste de paires de chaînes représentant des attributs est bien formée, c'est à dire que les premières composantes de chaque paire sont uniques. Par exemple :

```
verif_att_list [("a", "25"); ("id", "toto"); ("foo", "bar")];; (* true *)
verif_att_list [("a", "25"); ("id", "toto"); ("a", "bar")];; (* false *)
```

Dans le deuxième cas, le nom d'attribut `"a"` apparaît deux fois, la fonction doit donc renvoyer **`false`**.

Réponse: On utilise l'algorithme vu en cours de suppression des doublons dans une liste triée, légèrement adapté :

```
    let verif_att_list l =
      let rec sans_doublon l =
         match l with
         [] | [ _ ] -> true (* la liste vide ou a un élément
                               n'a pas de doublon *)
         | e1 :: e2 :: ll ->
                  if e1 = e2 then false (* on a trouvé un doublon *)
                  else sans_doublon (e2 :: ll)
                        (* on se rappelle sur le suivant.*)

    in
```

```
    (* important il faut trier la liste pour que ça marche *)
    sans_doublon (List.sort compare l)
```

6. (3 points) Écrire une fonction `valide : doc -> bool` qui renvoie **`true`** si et seulement si la valeur passée en argument vérifie les contraintes. On pourra utiliser une fonction récursive auxiliaire :

```
...
let rec val_aux doc pile =
   ...
in
...
```

l'argument `pile` est une liste de chaînes de caractères correspondant à des noms de balises rencontrés. En parcourant récursivement la liste `doc`, on inspecte le premier élément :

(a) en cas de balise ouvrante, on vérifie les attributs (avec la fonction précédente, question 5). S'ils sont incorrects, on renvoie **`false`**. Sinon, on ajoute le nom de la balise en tête de `pile`

(b) en cas de balise fermante, si `pile` est la liste vide, alors on renvoie **`false`**

(c) en cas de balise fermante, si le premier élément de `pile` est le même que celui de la balise, on le retire et on continue sur la suite du document, sinon on renvoie **`false`**

(d) en cas de texte, on vérifie la suite du document

(e) si le document est vide, si la `pile` est vide, on renvoie **`true`**, sinon on renvoie **`false`**.

Le cas (a) permet de détecter des attributs répétés. Il permet aussi de se souvenir du nom de la balise rencontrée en la mettant dans la `pile`. Le cas (b) permet de détecter des documents invalides tels que `<a></a></b>` (ici on empile `"a"`, puis on le dépile, puis on rencontre une balise fermante alors que l'on n'a pas empilé de `"b"` correspondant, erreur). Le cas (c) permet de détecter les balises mal refermées, par exemple `<a></b>`. Enfin, le cas (e) permet de vérifier qu'il ne reste pas des balise ouverte en suspens lorsque l'on a fini le document, par exemple `<a><b></b>`.

Réponse: On traduit vraiment l'énoncé en OCaml rien de plus. Il est important de noter que l'on est **sur le papier**. Donc une fonction qui a la bonne structure, même incomplète **rapporte des points**.

```

let valide doc =
   (* on se souvient qu'un doc est une liste *)

let rec val_aux l pile =
match l, pile with
BO(n, att) :: ll, _  -> (* cas (a) *)
                        if verif_att_list att then
                           val_aux ll (n :: pile)
                        else false
| BF _ :: _, []  ->     (* cas (b) *) false
| BF n1 :: ll, n2 :: pp -> (* cas (c) *)
                            if n1 = n2 then val_aux ll pp
                            else false
| T _ :: ll , _  -> val_aux ll pile
| [], [] -> true (* cas (e) *)
| [], _ -> false
in
val_aux doc []
```

Une astuce : quand on se rend compte que l'on doit tester deux choses (ici la liste ET la pile), alors il est souvent plus compact de faire **`match`** sur les deux choses, ce qui évite de refaire des **`match`** ou des **`if then else`** à droite des flèches (mais cette approche reste possible).

Aide-ménoire OCaml

Cet aide mémoire rappelle les types de bases en OCaml ainsi que les fonctions utilitaires associées ainsi que leurs types. Attention, toutes ces fonctions ne sont pas forcément utiles pour les exercices. Dans la suite, lorsqu'une fonction est marquée comme « opérateur binaire » (par exemple +) cela signifie qu'il faut l'écrire `a op b`. Sinon c'est une fonction qu'il faut appeler avec `op a b`.

Entiers

Le type `int` représente des entiers signés. Les constantes entières s'écrivent simplement `0`, `42`, `-233`. Les opérations et fonctions sur les entiers sont :

`+ : int -> int -> int` addition entre deux entiers (opérateur binaire).
`- : int -> int -> int` soustraction entre deux entiers (opérateur binaire).
`* : int -> int -> int` multiplication entre deux entiers (opérateur binaire).
`/ : int -> int -> int` division **entière** entre deux entiers (opérateur binaire).
`mod : int -> int -> int` reste dans la division entière (opérateur binaire).
`int_of_string : string -> int` conversion d'une chaîne en entier. Lève une exception si la chaîne n'est pas au bon format.
`int_of_float : float -> int` conversion d'un flottant en entier (la partie décimale est tronquée).

Flottants

Le type `float` représente des nombre flottants. Les constantes flottantes s'écrivent en notation scientifique `0.5`, `42e3`, `-233.8e-20`. Les opérations et fonctions sur les flottants sont :

`+. : float -> float -> float` addition entre deux flottants (opérateur binaire).
`-. : float -> float -> float` soustraction entre deux flottants (opérateur binaire)
`*. : float -> float -> float` multiplication entre deux flottants (opérateur binaire)
`/. : float -> float -> float` division entre deux flottants (opérateur binaire)
`** : float -> float -> float` puissance entre deux flottants (opérateur binaire)
`float_of_string : string -> float` conversion d'une chaîne en flottant. Lève une exception si la chaîne n'est pas au bon format.
`float : int -> float` conversion d'un flottant en entier (la partie décimale est tronquée).
`sqrt : float -> float` racine carrée d'un flottant.

Booléens

Le type `bool` représente des booléens. Les constantes booléennes sont `true` et `false`. Les opérations et fonctions sur les booléens sont :

`&& : bool -> bool -> bool` « et » logique entre deux booléens (opérateur binaire).
`|| : bool -> bool -> bool` « ou » logique entre deux booléens (opérateur binaire).
`not : bool -> bool` négation d'un booléen.

Chaînes de caractères

Le type `string` représente des chaînes de caractères. Les chaînes de caractères constantes sont délimitées par des guillemets :`"Hello, world !"`. De façon usuelle, la séquence d'échappement `\n` représente un retour à la ligne. Les opérations et fonctions sur les chaînes sont :

`^ : string -> string -> string` concaténation entre deux chaînes (opérateur binaire).
`String.length : string -> int` longueur d'une chaîne.
`String.trim : string -> string` renvoie une copie de la chaîne où les blancs (espaces, tabulations, retours à la ligne) en début et en fin de chaîne ont été supprimés.

Affichage

La fonction `Printf.printf fmt arg1 arg2 ... argn`, permet d'afficher `n` arguments en utilisant la chaîne de format `fmt`. Cette dernière est une chaîne de caractères contenant des séquences spéciales :
`%d` Affichage d'un entier.
`%s` Affichage d'une chaîne.
`%f` Affichage d'un flottant.

Exemple : `Printf.printf "Salut, mon nom est %s et j'ai %d ans""Toto"42` affiche :

```
Salut, mon nom est Toto et j'ai 42 ans
```

Comparaisons

En OCaml les opérations de comparaison permettent de comparer n'importe quelles valeurs du même type :
`<, <=, >, >=, =, <> : 'a -> 'a -> bool` comparaisons entre deux valeurs (respectivement, inférieur, inférieur ou égal, supérieur, supérieur ou égal, égal et différent), (opérateur binaire).
`compare : 'a -> 'a -> int` comparaison générique : `compare x y` renvoie un entier négatif si `x < y`, nul si `x = y` et positif si `x > y`.
`min : 'a -> 'a -> 'a` renvoie la plus petite de deux valeurs du même type.
`max : 'a -> 'a -> 'a` renvoie la plus petite de deux valeurs du même type.

n-uplets

Les n-uplets ou produits sont délimités par des parenthèses et des virgules. Dans le cas particulier des paires, deux fonctions `fst` et `snd` permettent d'accéder à la première et seconde composante. Dans les autres cas, on peut utiliser un `let` multiple :

```
1  let p1 = (10, 12);;
2  let p2 = (-1, false);;
3  let t3 = ("A", "B", 24);;
4
5  let x = fst p1;; (* 10 *)
6  let y = snd p2;; (* false *)
7  let a, b, n = t3;;  (* a vaut "A", b vaut "B" et n vaut 24 *)
```

Définitions de types

La directive `type t = ...` permet de définir un type OCaml nommé `t`. Ce type peut être :

Un produit nommé est défini par une expression de type donnant pour chaque étiquette le type des valeurs associées :

```
1  type point_colore = { x : float; y : float; couleur : string }
```

On peut créer des valeurs de ces types avec des accolades et accéder aux champs avec la notation `.f` :

```
1  let prouge = { x = 0.5; y = 10.3; couleur = "rouge" } ;;
2
3  (* Fonction pour afficher un point coloré *)
4  let pr_point p = Printf.printf "<x = %f, y = %f, couleur = %s>"
5         p.x p.y p.couleur;;
```

Un type somme est défini par une expression de type donnant la liste de cas possibles :

```
1  type val_carte = Roi | Dame | Valet | Val of int
```

On peut créer des valeurs de ces types en utilisant les constantes ou en leur donnant un argument. On peut tester les valeurs en utilisant l'opérateur de filtrage :

```
let dix = Val 10;;
let as = Val 1;;

(* Fonction pour afficher une valeur de carte *)
let pr_val v = match v with
     Roi -> Printf.printf "%s"  "Roi"
   | Dame -> Printf.printf "%s"  "Dame"
   | Valet -> Printf.printf "%s"  "Valet"
   | Val (1) -> Printf.printf "%s"  "As"
   | Val (n) -> Printf.printf "%d"  n;;
```

Exceptions

En OCaml, les exceptions sont des objets particuliers permettant de signaler une erreur. On peut définir une exception avec la directive **exception E of ...** :

```
exception MonErreur of string (* un message *)
```

On peut « lever » une exception, c'est à dire signaler une erreur au moyen de la fonction prédéfinie **raise** :

```
let err_arg_invalide () = raise (MonErreur "argument invalide");;
```

Une exception non rattrapée interrompt immédiatement le programme. On peut rattraper une exception avec la construction **try with** :

```
try
  18 + (f 42)  (* f peut lever une exception *)
with
   Not_found -> (* si l'exception Not_found est levée par f *)
                10
 | MonErreur msg ->
                12
```

Si on veut signaler une erreur avec un message, la fonction prédéfinie **failwith** permet de lever une exception avec ce message en argument.

```
if x < 0.0 then
     failwith "valeur négative interdite"
else
    sqrt x;;
```

Listes

Le type OCaml des listes est `'a list` et permet de représenter une collection ordonnée de valeurs du type `'a`. Les listes constantes sont délimitées par des crochets et des points-virgules : `[1; 2; 3; 10; 42; -5]`. La liste vide est représentée par `[]`. Les opérations et fonctions sur les listes sont :

`:: : 'a -> 'a list -> 'a list` ajout en tête de liste : `1 :: l` (opérateur binaire).

`@ : 'a list -> 'a list -> 'a list` concaténation de deux listes.

`List.iter : ('a -> unit) -> 'a list -> unit` `List.iter f l` applique `f` à tous les éléments de `l`. La fonction `f` ne renvoie pas de résultat (par exemple elle fait un affichage).

`List.map : ('a -> 'b) -> 'a list -> 'b list` `List.map f l` applique `f` à tous les éléments de `l` et renvoie la liste des images par `f`.

`List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a List.fold_left f init l` applique la fonction de combinaison `f` à `init` et tous les éléments de `l` dans l'ordre. Si `l = [ v1; v2; ... ; vn]`, alors `List.fold_left f init l = (f ... (f (init v1) v2) ... vn)`

`List.filter : ('a -> bool) -> 'a list -> 'a list List.filter f l` renvoie la liste de tous les éléments de `l` pour lesquels `f` renvoie **`true`**.

`List.assoc : 'a -> ('a * 'b) list -> 'b List.assoc a l` renvoie la seconde composante de la première paire dans `l` qui possède `a` comme première composante. La fonction lève l'exception `Not_found` si une telle paire n'existe pas.

`List.sort : ('a -> 'a -> int) -> 'a list -> 'a list  List.sort f l` renvoie une copie triée de `l` selon la fonction de comparaison `f`. Cette dernière suit les mêmes conventions que la fonction prédéfinie `compare`.

Enfin, on peut inspecter les listes au moyen de l'opérateur de filtrage :

```
1     (* teste si une liste est de longueur paire *)
2   let rec liste_long_paire l =
3     match l with
4     []    -> true (* la liste vide est de longueur 0, pair *)
5   | [ _ ] -> false (* la liste a un élément est de longueur 1, imapair *)
6   | _ :: _ :: ll -> liste_long_paire ll (* cas récursif *)
7   ;;
```