TP3 - Listes et récursion sur les listes en OCaml

Consignes générales

Ce TP3 permet de travailler les dernières notions vues en Cours (récursion et structure de données liste) qui seront au programme du Contrôle Partiel PFITA.

Cette feuille est composée de trois parties indépendantes réalisables dans un ordre quelconque :

- **La partie A** propose plusieurs exercices classiques sur les listes, complémentaire des exercices vus en Cours.
- **La partie B** propose plusieurs exercices brefs (petites fonctions à écrire ou expressions à typer) pour réviser les opérations élémentaires réalisables sur les listes.
- **La partie C** correspond aux exercices normalement vus au Cours/TD 6 (résolution du problèmes des Tours de Hanoï grâce à la récursivité, définition récursive de deux algorithmes de tris, et diverses fonctions de manipulation des listes).

Note : les énoncés notés avec une étoile comme **1)*** ne peuvent pas être testés par la plateforme. Il est de votre ressort de mener toutes les vérifications nécessaires, notamment avec l'aide du *toplevel*.

De plus, étant donné la longueur relative du TP et de la suite de tests associés, la fonctionnalité 'Noter!' prendra un temps non négligeable, donc n'hésitez pas à privilégier les fonctionnalités `Compiler` et `Évaluer le code` pour tester vos fonctions de manière presque autonome… avant de `Noter` vos exercices, idéalement justes du premier coup ! :)

Enfin, n'hésitez pas à solliciter votre encadrant de TP si vous avez des questions.

Partie A - Exercices "classiques" sur les listes

Exercice A.1 - Fonctions d'ordre supérieur utiles pour la suite

Écrire les fonctions suivantes (*sans utiliser la bibliothèque standard* `List` *qui contient déjà ces fonctions* :)

1) `filter` qui, prenant en argument **un prédicat unaire et une liste**, construit la liste des éléments vérifiant le prédicat.

Rappel : un "prédicat" est une fonction renvoyant un booléen, et l'adjectif "unaire" précise que cette fonction n'a qu'un élément.

2) `forall` qui, prenant en argument **un prédicat unaire et une liste**, renvoie vrai si et seulement si tous les éléments de la liste vérifient le prédicat.

3) `exists` qui, prenant en argument **un prédicat unaire et une liste**, renvoie vrai si et seulement s'il existe au moins un élément de la liste vérifiant le prédicat.

Exercice A.2 - Représentation d'une fonction unaire

Note : cet exercice A.2 correspond au sujet qui avait été posé au DM1 PFITA en 2016.

On considère des fonctions unaires qui sont représentées uniquement sur un domaine fini, par une liste de couples donnant pour chaque valeur de `x` (dans ce domaine fini) son image par la fonction.

Par exemple, la fonction `f` sur l'intervalle `[1,5]` t.q. `f(x) = 2x + 1` est représentée par la liste de couples :

```
[(1,3) ; (2,5) ; (3,7) ; (4,9) ; (5,11)]
```

Notez que l'**ordre des couples dans une telle liste est indifférent** car la notion mathématique sous-jacente est juste un *ensemble de couples* (donc les listes considérées ne seront pas forcément "triées" dans cet exercice).

Notez également que, pour representer correctement une fonction, les premiers éléments des couples, doivent être tous différents.

0) Écrire les fonctions `fst` et `snd` renvoyant la première composante (resp. la seconde composante) d'un couple.

1) Écrire une fonction `estFonction` qui, étant donnée une liste de couples, vérifie que la liste des premiers éléments ne contient pas de duplication.

On suppose maintenant qu'on a bien des fonctions.

2) Écrire la fonction `image` qui, étant donnés un élément et une liste de couples (représentant `f`) renvoie la valeur associée à l'élément si elle existe. Une exception sera levée avec `failwith` si l'élément n'a pas d'image.

3) Écrire la fonction `imageEns` qui, étant données une liste d'éléments `l` et une liste de couples (représentant une fonction `f`) renvoie la liste des valeurs associées à chaque élément de `l`. On supposera, sans le vérifier, que les éléments de la liste `f` sont bien dans le "domaine de définition" de `f`.

Indication : vous pouvez utiliser la fonction précédente.

4) Écrire `estInjective` qui appliquée à une fonction `f` représentée par une liste de couples vérifie que deux éléments n'ont pas la même image.

5) Écrire la fonction `surcharge` prenant 2 listes représentant les fonctions `f1` et `f2` et renvoyant une liste représentant la fonction `f` définie sur l'union des domaines de définition de `f1` et `f2` et dont l'image d'un élément est donnée soit par `f1`, soit par `f2` en donnant priorité à `f2`.

6) Écrire la fonction `composition` prenant 2 listes représentant les fonctions `f1` et `f2` et renvoyant une liste représentant la fonction `f` dont l'image d'un élément est donnée par `f1` appliquée à l'image par `f2` de cet élément.
Cette fonction correspond à l'opération de composition de fonctions `f1 o f2`, et *ne devra pas lever d'exception* si `"f2(x)"` n'a pas d'image par la fonction `f1`. En effet dans ce cas de figure, il n'y a "pas d'erreur" d'un point de vue mathématique : c'est juste que la fonction composée `f1 o f2` n'est pas définie en `x`.

Indication : définir une fonction auxiliaire `isDef` indiquant si un point `x` possède une image par une fonction `f`.

7) Écrire la fonction `produit` prenant en argument deux listes représentant deux fonctions `f1` et `f2` et renvoyant la liste représentant la fonction qui à un couple `(x,y)` associe le couple `(f1 x, f2 y)`.

Partie B - Exercices supplémentaires sur les listes

Exercice B.1 - Construction et manipulation de listes

1)* Construire la liste à 3 éléments contenant les entiers 10, 20 et 30 en utilisant d'une part `::` et `[]` et d'autre part `[`, `]` et le séparateur `;`.

2) Écrire les fonctions `head` et `tail`, qui, étant donnée une liste non vide, renvoie respectivement le 1er élément de la liste et la liste privée de son premier élément. On lèvera une exception avec `failwith` dans le cas d'une liste vide.

3)* Composer les fonctions `head` et `tail` pour accéder, s'il existe, au 3ème élément de chacune des listes suivantes :

```
[1;2;3;4]    [(1,2);(3,4);(5,6)]    []
[1;2]        [[1];[2;3;4];[]]       [[1,2];[3,4];[5,6]]
```

Exercice (B.2)* - Typage

Donnez le type, s'il existe, des expressions suivantes :

1) `[]`

2) `[1;2;true]`

3) `[1;(2,true)]`

4) `[1,2,3]`

5) `[[1,2];[3,4]]`

6) `[[1,2];[3,4,5]]`

7) `[1;2;3]`

8) `[(1,true,5.0);(2,false,6.4);(3,true,7.9)]`

9) `([1;2;3],[[];[true,false]])`

Exercice B.3 - Une fonction simple manipulant des listes

Écrire la fonction `consCpleDouble` qui, étant donné un entier `n`, construit la liste des couples `(i,2i)` pour `i = 1` à `n`.

Remarque 1 : L'ordre des couples dans la liste résultat n'a pas d'importance.

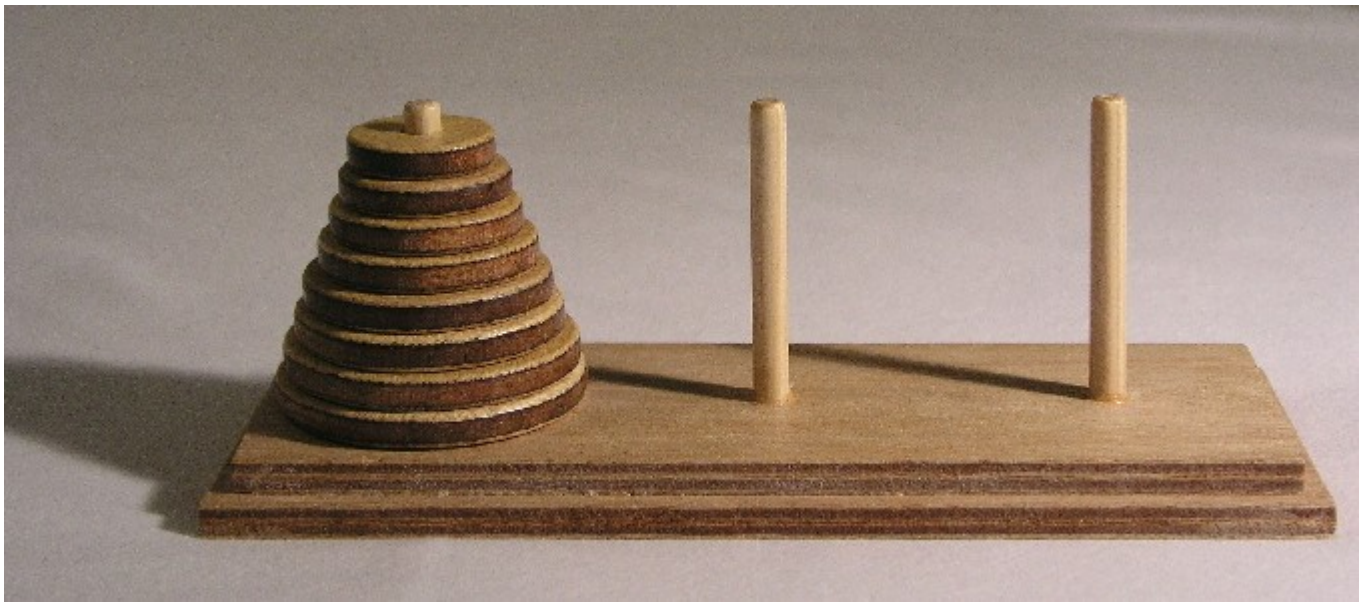
Remarque 2 : la fonction renverra la liste vide en cas d'argument `n <= 0`.

Partie C - Retour sur le Cours/TD 6

Exercice C.1 - Tours de Hanoï

Le problème des tours de Hanoï est un jeu de réflexion imaginé par le mathématicien français Édouard Lucas. C'est un très bon exemple pour illustrer l'expressivité de la récursivité (s'agissant ici, de la récursivité sur les entiers : les listes ne seront utilisées qu'en résultat).

Le but du jeu est de déplacer des disques de diamètres différents d'une tour de départ (Tour 1) à une tour de destination (Tour 3) en passant par une tour intermédiaire (Tour 2).



Les règles de jeu sont les suivantes :

- Ne déplacer qu'un disque à la fois.
- Un disque ne peut être déplacé que si l'emplacement est vide ou si le disque de réception est plus grand.

En raisonnant de manière inductive, écrire la fonction `hanoi` qui prend en argument le nombre `n` de disques ainsi qu'un triplet `(source, temp, dest)` de 3 paramètres supposés différents (qui identifient les tours), et renvoie la liste des mouvements définis par des couples

`(identifiant de la tour de départ, identifiant de la tour d'arrivée).`

Votre fonction devra avoir le type

```
hanoi : int -> 'a * 'a * 'a -> ('a * 'a) list
```

Indication : Voici le résultat de `hanoi` sur un exemple :

```
hanoi 2 (1,2,3) = [(1, 2); (1, 3); (2, 3)]
```

En cas d'argument non valide (p.ex. `hanoi (-1) (1,2,3)`), votre fonction devra juste renvoyer la **liste vide**.

Exercice C.2 - Tri

Dans cet exercice deux algorithmes de tri sont a réaliser. Le tri doit être croissant et porte sur une liste d'entiers.

1) *Tri par insertion :* Écrire la fonction `inserer` qui, étant donné un entier et une liste déjà triée, construit la liste où l'entier est inséré en bonne position dans la liste. En déduire une fonction `triInsertion` qui trie une liste d'entiers dans l'ordre croissant.

2) *Tri par fusion :* Écrire la fonction `partage` qui partage une liste en deux listes `l1` et `l2` telles que les tailles de `l1` et `l2` ne diffèrent que d'un au maximum. Écrire une fonction `merge` qui prend en argument deux listes ordonnées d'entiers et renvoie une liste ordonnée (vous ne devrez pas utiliser la fonction `inserer` pour cette fonction). En déduire une fonction `triFusion` qui trie une liste d'entiers dans l'ordre croissant.

Exercice C.3 - Écrire les fonctions suivantes

1) `last` qui renvoie le dernier élément d'une liste, s'il existe.

2) `sum` qui renvoie la somme des éléments d'une liste d'entiers.

3) `append` qui, prenant en argument deux listes, concatène les deux listes. Cette fonction est déjà définie dans la bibliothèque standard `List`, mais on se propose de la redéfinir à la main pour s'entraîner.

4) `reverse` qui, prenant en argument une liste, construit la liste en inversant l'ordre les éléments. **Remarque :** cette fonction admet plusieurs implémentations possibles. Vous utiliserez *celle qui admet l'écriture la plus simple et la plus proche du style fonctionnel* (et vous aurez le droit d'utiliser l'opérateur de concaténation `( @ )`, qui correspond à la fonction `List.append` dans la bibliothèque standard).

5) `nbOcc` qui, prenant en argument un élément et une liste, compte le nombre d'occurrences de l'élément dans la liste.

6) `elimFirst` qui, prenant en argument un élément et une liste, construit la liste où la 1ère occurrence de l'élément a été supprimée.

7) `elimLast1` (et `elimLast2`) qui, prenant en argument un élément et une liste, construit la liste où la dernière occurrence de l'élément a été supprimée. Vous implémenterez deux variantes pour cette fonction : avec une implémentation `let rec` directe, et sans `let rec` mais en utilisant les fonctions précédentes (`reverse`…)

8) `elim` qui, prenant en argument un élément et une liste, construit la liste où toutes les occurrences de l'élément ont été supprimées.

9) `substitute` qui, prenant en argument 2 éléments `x` et `y` et une liste, construit la liste où toutes les occurrences de `x` sont remplacées par `y`.

10) `substitute2` qui, prenant en argument un prédicat unaire `p`, un élément et une liste, construit la liste où tous les éléments de la liste vérifiant le prédicat `p` sont remplacés par l'élément donné.