TP 2 : fonctions et listes

Consigne 1 : spécifier le type des paramètres et du résultat des fonctions direcement dans la déclaration des fonctions, comme on l'a vu en cours

Consigne 2 : une fois le code écrit et compilé sans erreur, il est bien sûr évident que vous le testerez sur quelques exemples. Vous choisirez évidemment ces exemples pertinement, de manière à tester les cas limites en particulier.

Conseil 1 : pensez récursivement !
Conseil 2 : vous pourrez utiliser l'opérateur `@` de concaténation des listes. On rappelle qu'il est de complexité linéaire par rapport à la première liste.
Conseil 3 : Vous pourrez utiliser également la fonction `List.mem` de type `'a -> 'a list -> bool` qui permet de déterminer si un élément appartient à une liste. Elle est de complexité linéaire en la taille de la liste dans le pire des cas.

I Ensembles

On représente un ensemble (fini) d'objets par la liste de ses éléments. La liste vide représente donc l'ensemble vide. Une liste représente un ensemble si et seulement si elle est sans doublons, i.e. les éléments n'apparaissent qu'une seule fois dans la liste.

1. Écrire une fonction `valide` de type `'a list -> bool` qui calcule le booléen indiquant si une liste représente effectivement un ensemble. Quelle est sa complexité ?
2. Écrire une fonction `unique`, qui supprime les doublons d'une liste : `unique [1; 2; 1; 0; 3; 2]` calcule la liste `[1; 2; 0; 3]` ou toute autre liste égale à celle-ci à une permutation près. Quelle est sa complexité ?
3. Écrire les fonctions `union`, `intersection` et `difference` qui opérent sur les ensembles (on pourra bien sûr réutiliser la fonction `unique` précédente si nécessaire). Estimer leurs complexités.
4. Écrire une fonction `inclus` qui calcule le booléen indiquant si un ensemble est inclus dans un autre.
5. Écrire la fonction `egal` qui calcule le booléen indiquant si deux ensembles sont égaux.
6. Un peu plus difficile ! Écrire une fonction `parties` qui calcule l'ensemble des parties d'un ensemble : `parties [0; 1; 2]` calcule `[[]; [2]; [1]; [1; 2]; [0]; [0; 2]; [0; 1]; [0; 1; 2]]` ou toute autre liste à une permutation près. La difficulté réside essentiellement dans la formulation de la relation de récurrence.

   Pour la programmation on pourra programmer une fonction utilitaire.

   Estimer sa complexité.

II Fibonacci et généralisation

On définit la célèbre suite $(f_n)_{n\in\mathbb{N}}$ de FIBONACCI par $f_0 = 0$, $f_1 = 1$, et $\forall n \geqslant 2, f_n = f_{n-1} + f_{n-2}$.

1. Écrire une fonction naïve `fibo` telle que `fibo n` calcule le n-ième terme de la suite de FIBONACCI.
2. Que dire a priori de sa complexité (ne pas montrer le résultat, mais penser aux tours de Hanoï...) ? On verra sous peu des façons plus efficaces d'écrire cette fonction.

   On s'intéresse à une fonction plus générique `fibogen` telle que `fibogen f0 f1 op n` retourne le n-ième élément de la suite $(u_n)_{n\in\mathbb{N}}$ définie par la relation suivante, où $*$ représente l'opérateur op : $u_0 = 0$, $u_1 = 1$, et $\forall n \geqslant 2, u_n = u_{n-1} * u_{n-2}$.

   L'opérateur `op` qui sera passé sera sous forme prefixe, i.e. pour l'appliquer à deux arguments `a` et `b`, on écrit `op a b`.

   Pour transformer un opérateur infixe déjà existant en opérateur préfixe il faut en OCAML l'entourer de deux parenthèses. Par exemple `( + )` est l'opérateur préfixe associé à l'addition. Dès lors `( + ) 3 5` renvoie bien 8.
3. Quel est le type de `fibogen` ?
4. Écrire la fonction correspondante.
5. Vérifiez pour quelques valeurs de n que `fibogen 0 1 ( + )` est bien la même fonction que la fonction `fibo` que vous avez écrite plus haut.
6. Quels sont les paramètres à donner à la fonction `fibogen` pour obtenir les fonctions `fiboword` et `fibolist` correspondant aux suites suivantes ?

   $w_0 = b$, $w_1 = a$, $w_2 = ab$, $w_3 = aba$, $w_4 = abaab$, $w_5 = abaababa$, ...

   et

   $\ell_0 = [1]$, $\ell_1 = [0]$, $\ell_2 = [0;1]$, $\ell_3 = [0;1;0]$, $\ell_4 = [0;1;0;0;1]$, $\ell_5 = [0;1;0;0;1;0;1;0]$, ...
7. En utilisant le mécanisme des fonctions partielles définir ces deux fonctions `fiboword` et `fibolist`
8. Écrire une fonction `nb_occ` qui détermine le nombre d'occurrences d'un élément d'une liste. Vérifier pour quelques valeurs de n que `nb_occ 0 figolist n = fibo n` (ne pas démontrer cette propriété !).
9. Écrire une fonction `tronque2` telle que `tronque2 liste` retourne la liste `liste` privée de ses 2 derniers éléments (on lèvera une exception "Liste trop courte" si nécessaire). Vérifier pour quelques valeurs de n, de préférence en vous aidant de `List.map`, que ℓ_n privée de ses 2 derniers éléments est toujours un palindrome (i.e. on obtient la même suite d'éléments en parcourant la liste du début à la fin ou de la fin vers le début). Il faudra donc écrire une fonction qui teste si une liste est un palidrome.
10. Écrire une fonction remplace, de type `'a -> 'a list ->'a list`, telle que `remplace x lx l` remplace chaque occurrence de `x` dans `l` par ceux de la liste `lx`. Par exemple `remplace 2 [1; 3] [0; 1; 2; 1; 2]` calcule la liste `[0; 1; 1; 3; 1; 1; 3]`.

    Vérifier pour quelques valeurs de n qu'en remplaçant tous les 0 par `[0;1]` et **en même temps**[1] tous les 1 par `[0]` dans ℓ_n, on obtient ℓ_{n+1} ! On pourra écrire une fonction `next` telle que `next l` calcule la nouvelle liste dans laquelle les deux substitutions ont été réalisées en même temps.

1. Il ne faut donc pas écrire `remplace 1 [0] (remplace 0 [0;1] l)`.