TP4 - Types utilisateur en OCaml

Consignes générales

Cette feuille est composée de deux parties indépendantes :

- **La partie A** est une brève introduction à la manipulation d'expressions définies à partir d'un type utilisateur ;
- **La partie B** se place dans un contexte plus large en manipulant plusieurs types utilisateur pour mobiliser vos connaissances acquises dans les thèmes précédents.

Nous vous recommandons de traiter ces deux parties du TP4 dans l'ordre de l'énoncé.

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

Avant de commencer la séance, assurez-vous que vous avez été noté présent pour cette séance de TP en présentiel. Et n'hésitez pas à solliciter votre encadrant de TP pour toute question.

À la fin de la séance, n'oubliez pas de sauvegarder votre travail en cliquant sur `Sync` ou `Noter` (voire en téléchargeant le code `.ml`).

Partie A - Implantation des listes comme un type utilisateur

Le TP précédent portait sur la construction et la manipulation de listes en utilisant la syntaxe dédiée d'OCaml (constructeurs `[]` et `_ :: _`), le filtrage et la récursion.

La construction et le filtrage sur les types utilisateur suit les mêmes idées, avec comme seule différence l'écriture des constructeurs : un identificateur commençant par une **Majuscule** et comprenant zéro, un argument, ou bien un *n*-uplet d'arguments. (On peut voir les constructeurs comme des "fonctions spéciales", qui **ne peuvent pas être curryfiées**.) Par exemple :

```
type nombre = Erreur | Reel of float | Cplx of float * float

let ex_reel = Reel 5.0;;
(* val ex_reel : nombre = Reel 5. *)
let ex_cplx = Cplx(1.0, 0.0);;
(* val ex_cplx : nombre = Cplx (1., 0.) *)
let ex_err = Erreur;;
(* val ex_err : nombre = Erreur *)
```

Le but de cet exercice est de montrer qu'il est possible de ré-implanter le type des listes avec la notion plus générale de type utilisateur vue dans cette partie du cours. Vous devrez ainsi écrire quelques fonctions récursives sur un tel type.

Considérons le type suivant (vous n'avez pas à le re-saisir dans votre code) :

```
type 'a liste = Nil | Cons of 'a * 'a liste
```

1) Définissez les expressions `liste_12` et `liste_liste0` correspondant respectivement à une liste comportant les deux éléments 1 et 2, et à une liste comportant une liste vide comme seul élément. (Il s'agit donc de l'analogue de `[1; 2]` et `[[]]`.)

2) Écrivez la fonction `dernier : 'a liste -> 'a` renvoyant le dernier élément d'une liste, s'il existe (et levant une exception sinon).

3) Écrivez la fonction `append : 'a liste -> 'a liste -> 'a liste` renvoyant la concaténation des deux listes en argument en préservant leur ordre.

Partie B - Arithmétique et Mobiles

Exercice B.1 - Arithmétique modulo 3

On considère l'arithmétique modulo 3 sur l'ensemble `m3` = {*Zero*, *Un*, *Deux*} défini par :

```
type m3 = Zero | Un | Deux
```

(vous n'avez pas à le re-saisir dans votre code).

1) Écrire la fonction `m3s` qui calcule le successeur modulo 3 dans `m3`.

2) Écrire la fonction `m3p` qui calcule le précécesseur modulo 3 dans `m3`.

3) Écrire la fonction `m3plus` qui définit l'opération d'addition modulo 3 dans `m3`.

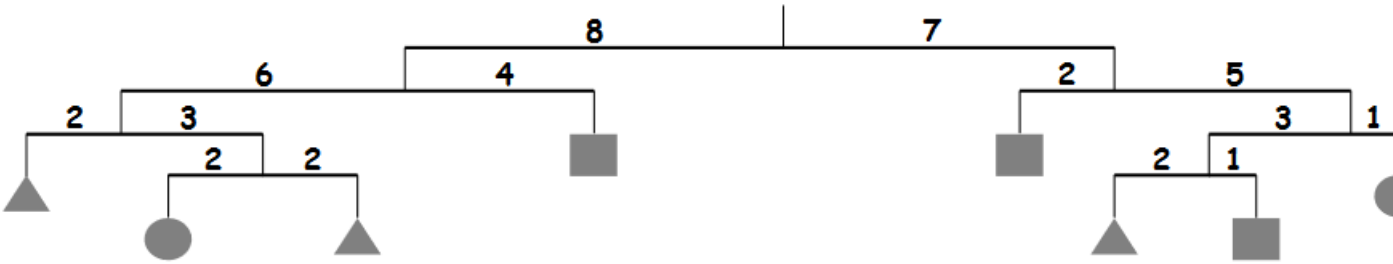
4) Écrire la fonction `m3mult` qui définit l'opération de multiplication modulo 3 dans `m3`.

5)* Une expression de type `exp` est définie soit comme une *constante* (typée par `m3`), soit la *somme*, soit le *produit* de 2 expressions de type `exp`. Définir le type `exp`.

6) Écrire la fonction `calculer` qui retourne la valeur dans {*Zero*, *Un*, *Deux*} d'une expression de type `exp`.
Définir deux expressions différentes `e1` et `e2` de type `exp` représentant la même valeur *Deux*.

Exercice B.2 - Mobiles

On considère des mobiles tels que le mobile `m0` représenté sur la figure ci-dessous.

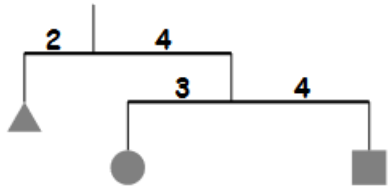


Les *entiers* représentent les longueurs des tiges (en nombre d'unités de longueur).

Les `figure`s du mobile sont les objets suspendus aux extrémités des tiges. Il y a 3 catégories de `figures` : des *cubes* caractérisés par la longueur de leur côté, des *sphères* caractérisées par leur rayon, des *pyramides* carrées caractérisées par leur hauteur supposée égale au côté de leur base. Un mobile est soit une *figure*, soit une *tige* caractérisée par la longueur d'un côté, le mobile accroché à son extrémité, la longueur de l'autre côté et le mobile de l'autre côté.

1)* Définir les types `figure` et `mobile`.

2) Donner la représentation OCaml du mobile *m1* ci-dessous sachant que le paramètre de taille des figures est 1.



3) Donner la représentation OCaml du mobile *m2* constitué uniquement d'un cube de taille 2.

4) Écrire la fonction `nbSphere` qui, étant donné un `mobile`, calcule le nombre de sphères apparaissant dans le `mobile`.

5) On appelle *hauteur* d'un mobile son nombre de niveaux, c'est à dire le nombre maximum de «barres» verticales rencontrées pour atteindre une feuille en partant du sommet du mobile. Ainsi, les mobiles *m0* et *m1* ont respectivement pour hauteur 5 et 3.

Écrire une fonction `hauteur` qui, étant donné un mobile, calcule sa hauteur.

6)* Écrire la fonction `echanger` qui, étant donnés un `mobile` et deux `figures` f1 et f2, construit un `mobile` en remplaçant partout dans le `mobile` la `figure` f1 par la `figure` f2.

Plus précisément, si vous avez appelé vos constructeurs de `figure` "`Cube`" et "`Sphere`" : `echanger m (Cube(1)) (Sphere(3))` devra remplacer récursivement dans `m`, toutes les occurrences de `Cube(1)` par `Sphere(3)` (mais ne pas impacter `Cube(2)` par exemple).

7)* Écrire la fonction `listeFigure` qui, étant donné un mobile, construit la liste de toutes les figures du mobile.

Remarque : On retournera une liste de type `figure list` (et non pas `figure liste`).

À chaque `figure`, on associe un poids proportionnel à son volume. On considère que :

- l'unité de longueur par défaut est le centimètre,
- la masse volumique d'une figure est de 9g/cm3,
- la masse des fils et des tiges est négligeable.

8)* Écrire les fonctions `masseFigure : figure -> float` et `masseMobile : mobile -> float` qui calculent respectivement la masse d'une figure et d'un mobile, en grammes.

Remarque : on utilisera la fonction `float` (a.k.a. `float_of_int`) pour convertir un nombre en entier en nombre flottant, et le calcul suivant permettra d'obtenir une approximation du nombre pi :

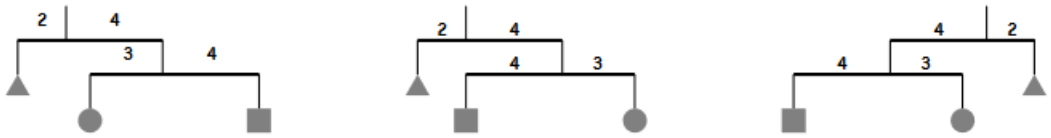
```
let pi = 4. *. atan 1.
```

9) Un mobile est dit localement équilibré s'il est réduit à une figure, ou si le produit de la distance du mobile de droite par sa masse est égal au produit de la distance du mobile de gauche par sa masse.
Écrire la fonction `equilibreLocal` qui détermine si un mobile est localement équilibré.

10) Un mobile est dit globalement équilibré si tous les mobiles qu'il contient sont localement équilibrés.
Écrire la fonction `equilibreGlobal` qui détermine si un mobile est globalement équilibré.

11)* On suppose maintenant que les tiges horizontales des mobiles peuvent pivoter de 180° par rapport à la base des tiges verticales. On dit que deux mobiles sont équivalents s'ils sont égaux ou si on peut les rendre égaux à la suite d'une série de rotations de 180° d'une ou plusieurs tiges horizontale

Exemple : les trois mobiles ci-dessous sont équivalents:



Écrire une fonction `mobilesEquiv`, qui, étant donnés deux mobiles, détermine s'ils sont équivalents.