TP n°3

1. Construire un type *nombre* permettant de représenter aussi bien des entiers, des réels et des complexes (définis comme des couples de réels). Définir alors une fonction qui calcule le produit de deux nombres quelconques pris dans cet ensemble de valeurs.

2. Représentation de listes "généralisées" de "profondeur" quelconque et ayant des éléments de types différents.

Une liste généralisée est

- soit un atome : une constante appelée Nil, ou un entier, ou un caractère.
- soit une liste de listes généralisées, qui sont les "composants" de la liste

Construire un type *Genlist* permettant de représenter de telles listes généralisées.

Etant donnée une liste généralisée L, on définit de manière inductive le "niveau" de constituants de cette liste:

- si L est réduite à un atome, celui-ci est de niveau 0, sinon les composants de L sont de niveau 1
- si une (sous-)liste de L est un composant de niveau *n* , ses composants ont le niveau *n+1* .

La profondeur d'une liste généralisée est alors définie comme le niveau le plus élevé de ses atomes.
Par exemple, si on utilise des parenthèses pour représenter une liste généralisée dans une syntaxe concrète plus agréable, la liste suivante a une profondeur de 5 (c'est le niveau de la deuxième occurrence de Nil) :

(5 ((1 (Nil) (0 X (Nil)) (Y))))

• Construire la fonction *profondeur* qui donne la profondeur d'une liste généralisée et testez la sur des exemples.

• Définir la fonction *substitue* qui, étant données trois listes généralisées *L, el1, el2*, permet de substituer toute occurrence de *el1* dans *L* par *el2*

• Définir une fonction qui permet d'afficher une liste généralisée, selon la syntaxe concrète parenthesée indiquée plus haut.

• Définir une fonction qui « met à plat » une liste généralisée *L*, c'est-à-dire donne une liste contenant les atomes de *L* en respectant leur ordre dans l'arborescence de *L*, et en supposant qu'une liste *L* réduite à un atome est laissée invariante.

3. Filtrages de termes algébriques

On considère le type *term* permettant de représenter des expressions algébriques avec opérateurs ayant un nombre quelconque d'arguments :

```
type term =   Term of  string * term list     (* string, pour le nom d'un opérateur *)
            | Var of string;;                 (* string, pour le nom d'une variable  *)
```

Une substitution est représentée par un liste associative de variables et termes:
[(v1, t1) ; (v2, t2) ; . . .(vn, tn)].
C'est en fait une fonction qui associe un t_i à chacune des variables v_i.
Appliquer une substitution *s* à un terme *t* signifie remplacer dans *t* chaque variable par le terme associé dans *s* (s'il y en a) ou sinon la laisser invariante.

• Définir la fonction *add_subst* qui rajoute un couple à un substitution, si cela est compatible: si le couple y est déjà, la substitution est inchangée, si la variable y est déjà, mais associée à un terme différent, c'est incompatible. Pour un échec, on utilisera l'exception *Match_exc* définie par :

```
exception Match_exc of string;;
```

• Définir la fonction *matching* qui permet de « filtrer » un terme quelconque *t* par un terme-filtre *f*.
Il s'agit en fait de trouver une substitution *s* telle que *s f = t*.
matching va « accumuler » des liaisons *(variable, terme)* dans une liste initialement vide en appliquant le principe suivant :

- une variable de *f* filtre n'importe quel (sous-)terme de *t*
- un noeud de *f* filtre un noeud de *t* si les opérateurs qui étiquettent les noeuds considérés sont les mêmes, sont utilisés avec le même nombre de fils, et si chaque fils filtre le fils correspondant.

On pourra utiliser la fonction *combine* qui transforme une paire de listes en liste de paires.