Arbres

Pour les calculs de complexité (à faire systématiquement à la volée), on comptera les (éventuels) appels récursifs, les matchings, et les constructions de structure avec le même poids.

Pour pouvoir utiliser la fonction `dessin_arbre` disponible dans les fichiers des 5/2, on utilisera dans les deux premiers exercices le type :

```
type 'a arbre_b=V|N of 'a*('a arbre_b)*('a arbre_b);;
```

Une feuille sera un nœud dont les deux fils sont vides.

Il est fortemement suggeré de définir des le début quelques arbres tests, dans une feuille de travail dédiée à ce TD, qu'on inclura sous Caml. Le `include "..."` sera d'ailleurs la seule chose qui apparaitra dans la fenetre de commande Caml, comme d'habitude...

Exercice 1 *Fonctions de base sur les arbres en Caml*

1. Ecrire des fonctions de base sur les arbres : `hauteur`, `nb_feuilles` et `nb_noeuds`.
2. Ecrire des fonctions créant des peignes (resp. arbres parfaits) d'une hauteur donnée. Dans les deux cas, on s'efforcera de distribuer des étiquettes de façon injective.
3. Ecrire une fonction donnant le miroir d'un arbre donné (au sens d'une symétrie par rapport à un "axe vertical").

Exercice 2 *Parcours d'arbre*

1. Ecrire des fonctions récursives retournant la liste des étiquettes d'un arbre, parcouru de façon infixe (resp. préfixe et postfixe).
2. Reprendre les fonctions précédentes de façon non récursive, en utilisant le type Pile suivant :

   ```
   type 'a pile==('a list) ref;;
   let Pile_vide ()= ref [];;
   let push x p=(p:=x:: !p);;
   let pop l=let x=hd !l in l:=tl !l;x;;
   let est_vide l=(!l=[]);;
   ```

   IMPORTANT : Prouver la terminaison et la validité de votre fonction. Vérifier que la complexité n'est pas dégradée.
3. La liste infixe (resp. préfixe, postfixe) des étiquettes d'un arbre est-elle suffisante pour décrire l'arbre ?
4. Donner un cadre raisonnable dans lequel la donnée des listes infixe et préfixe suffit à reconstituer l'arbre.
5. Dans le cadre précédent, écrire une fonction qui reconstitue un arbre à partir des listes infixe et préfixe de ses étiquettes.

Exercice 3 *Un problème d'analyse syntaxique*

On s'intéresse ici à des arbres d'arité quelconque, que l'on représentera en Caml par le type :

```
type arbre={Etiq:int; fils:arbre list};;
```

Le but du jeu est de représenter ces arbres de façon équivalente par des expressions parenthèsées : un arbre de racine étiquetée par e sera représentée par $(e(r_1)(r_2)\dots(r_k))$ où $r_1, \dots, r_k$ sont les représentations des fils de la racine.

Pour zapper la partie "analyse lexicale", on travaillera directement sur des listes de *lexèmes* : `type Lex=Val of int|PO|PF;;`

Par exemple, l'arbre défini en Caml par :

```
arbre2exp {Etiq=12; fils=[{Etiq=23;fils=[]};{Etiq=13;fils=[]}]};;
```

peut être décrit par l'expression parenthèsée (12(23)(13)), elle-même représentée par la liste de lexèmes :

```
[PO; Val 12; PO; Val 23; PF; PO; Val 13; PF; PF]
```

1. Ecrire une fonction calculant l'expression parenthèsée (donnée sous forme de liste de lexèmes) associée à un arbre.
2. Ecrire une fonction faisant le travail inverse.

   Dans les deux cas, on pourra traiter en parallèle les arbres et les listes d'arbres. Pour l'analyse syntaxique (fonction `exp2arbre`*), il est suggeré d'écrire des fonctions auxiliaires rendant un couple constitué de l'analyse d'un boût de liste, et de la liste restant à analyser*
3. Comment dérécursifier la fonction `exp2arbre` (décrire les structures de données, etc...)
4. Reprendre l'exercice 1 avec ce type d'arbres (pour les arbres complets, on prendra en paramètre l'arité des nœuds autres que les feuilles. Au passage, on calculera le nombre d'éléments de tels arbres.)