Corrigé feuille d'exercices n°2 de l'option d'informatique.

Exercice n°1

```
1  type arbre = noeud of arbre * arbre | feuille;;
2
3  (* fonction "hauteur" qui calcule la hauteur ou la profondeur d'un arbre *)
4  let rec hauteur = function
5      noeud(fg,fd) -> 1 + max (hauteur fg) (hauteur fd)
6    | feuille -> 0;;
7
8  (* fonction "nbnoeuds" qui calcule le nombre de noeuds dans un arbre donné *)
9  let rec nbnoeuds = function
10     noeud(fg,fd) -> 1 + (nbnoeuds fg) + (nbnoeuds fd)
11   | feuille -> 0;;
12
13 (* fonction "nbfeuilles" qui calcule le nombre de feuilles d'un arbre *)
14 let rec nbfeuilles = function
15     noeud(fg,fd) -> (nbfeuilles fg) + (nbfeuilles fd)
16   | feuille -> 1;;
```

Exercice n°2

Définition du type «`dico`» arbre dont les noeuds sont les mots du dictionnaire.

```
17 type dico =
18   noeud of dico * string * dico
19 | vide;;
20
21 (* création d'un arbre simple à 2 feuilles à partir d'un mot *)
22 let creearbre mot = noeud(vide,mot,vide);;
```

La fonction «`insere`» permet l'insertion de «mots» dans un arbre dictionnaire en respectant l'ordre lexicographique. Un mot déjà présent n'est pas rajouté.

```
23 let rec insere mot = function
24    noeud(fg,mot1,fd) as arbre when mot = mot1 -> arbre
25 | noeud(fg,mot1,fd) when mot1 > mot -> noeud(insere mot fg,mot1,fd)
26 | noeud(fg,mot1,fd)                              -> noeud(fg,mot1,insere mot fd)
27 |       vide                                              -> noeud(vide,mot,vide);;
```

La fonction «`arbre_de_liste`» transforme une liste en un arbre dictionnaire.

```
28 let rec arbre_de_liste = function
29    x::xs -> insere x (arbre_de_liste xs)
30  | [] -> vide;;
```

La fonction «lit» parcourt un arbre dictionnaire et affiche à l'écran les mots dans l'ordre lexicographique

```
31  let rec lit = function
32     noeud(fg,mot,fd) -> lit fg; print_string mot;
33                         print_newline ();lit fd
34   | vide -> ();;
```

La fonction «liste_de_arbre» transforme un arbre en liste de mots.

```
35  let rec liste_de_arbre = function
36     noeud(fg,mot,fd) -> (liste_de_arbre fg)@[mot]@(liste_de_arbre fd)
37   | vide -> [];;
```

La fonction «max_arbre» retourne le plus grand élément d'un arbre.

```
38  let rec max_arbre = function
39     noeud(fg,mot,vide) -> mot
40  | noeud(fg,mot,fd) -> max_arbre fd
41  | vide -> failwith "max_arbre : erreur, arbre vide";;
```

La fonction «supprime» élimine un mot dans un arbre.

```
42   let rec supprime mot = function
43     noeud(fg,mot1,fd) when mot1 > mot
44             -> noeud(supprime mot fg,mot1,fd)
45   | noeud(fg,mot1,fd) when mot1 < mot
46             -> noeud(fg,mot1,supprime mot fd)
47   | noeud(fg,mot1,vide) (* ici nécessairement mot1=mot *)
48             -> fg
49   | noeud(vide,mot1,fd) (* ici nécessairement mot1=mot *)
50             -> fd
51   | noeud(fg,mot1,fd) (* ici nécessairement mot1=mot *)
52             -> let mot2 = max_arbre fd in
53                 noeud(fg,mot2,supprime mot2 fd)
54   | vide -> print_string (mot^" non trouvé"); print_newline (); vide;;
```

Exercice n°3

```
55  type opelog = et | non | ou | impliq | equiv | xou | nonet;;
56
57  type constlog = vrai | faux;;
58
59  (* feuillec = feuille pour les constantes
60     feuillev = feuilles pour les variables
61     noeuds = noeud simple pour les fonctions
62     noeudd = noeud double pour les opérateurs *)
63  type arbrelog = feuillec of constlog | feuillev of string
64  | noeuds of opelog * arbrelog | noeudd of arbrelog * opelog * arbrelog;;
```

```
65  let rec evalexpr = function
66  feuillev(_) -> failwith "erreur : arbre non évaluable"
67    | feuillec(vrai) -> true
68    | feuillec(faux) -> false
69    | noeuds(non,fils) -> not(evalexpr fils)
70    | noeuds(_,_) -> failwith "erreur : expression erronée"
71    | noeudd(fg,et,fd) -> (evalexpr fg)&&(evalexpr fd)
72    | noeudd(fg,ou,fd) -> (evalexpr fg)||(evalexpr fd)
73    | noeudd(fg,_,fd) -> failwith "erreur : opérateur non traité"
74  ;;
```

Pour transformer un arbre logique quelconque en un arbre logique ne contenant que des «ou», des «et» et des «non».

```
75  let rec transforme = function
76    feuillec(a) as f -> f
77    | feuillev(c) as f -> f
78    | noeuds(non,fils) -> noeuds(non,transforme fils)
79    | noeuds(_,_) -> failwith "erreur : arbre non conforme"
80    | noeudd(fg,et,fd) -> noeudd(transforme fg,et,transforme fd)
81    | noeudd(fg,ou,fd) -> noeudd(transforme fg,ou,transforme fd)
82    | noeudd(fg,impliq,fd) -> let fg' = transforme fg and fd' = transforme fd in
83                        noeudd(noeuds(non,fg),ou,fd')
84    | noeudd(fg,equiv,fd) -> let fg' = transforme fg and fd' = transforme fd in
85          noeudd(noeudd(fg',et,fd'),ou,noeudd(noeuds(non,fg'),et,noeuds(non,fd')))
86    | noeudd(fg,xou,fd) -> let fg' = transforme fg and fd' = transforme fd in
87          noeudd(noeudd(noeuds(non,fg'),et,fd'),ou,noeudd(fg',et,noeuds(non,fd')))
88    | noeudd(fg,nonet,fd) -> let fg' = transforme fg and fd' = transforme fd in
89                noeudd(noeuds(non,fg'),ou,noeuds(non,fd'))
90    | noeudd(_,_,_) -> failwith "erreur : arbre non conforme"
91  ;;
```

`evalexpr` est une fonction qui permet d'évaluer une expression logique donnée sous forme d'arbre.

```
92  let rec evalexpr = function
93  feuillev(_) -> failwith "erreur : arbre non évaluable"
94    | feuillec(vrai) -> true
95    | feuillec(faux) -> false
96    | noeuds(non,fils) -> not(evalexpr fils)
97    | noeuds(_,_) -> failwith "erreur : expression erronée"
98    | noeudd(fg,et,fd) -> (evalexpr fg)&&(evalexpr fd)
99    | noeudd(fg,ou,fd) -> (evalexpr fg)||(evalexpr fd)
100   | noeudd(fg,_,fd) -> failwith "erreur : opérateur non traité"
101 ;;
```

Pour écrire une fonction qui récupère toutes les variables d'un arbre logique, on insère les variables dans une liste triée au fur et à mesure du parcours de l'arbre.

```
(* fonction d'insertion dans une liste triée *)
let rec insere x = function
  [] -> [x]
 | y::ys when x<y -> x::y::ys
 | y::ys when x=y -> y::ys
 | y::ys (* when x>y *) -> y::(insere x ys);;

let rec liste_var arbre =
 let rec aux = fun
     l (feuillec(a))     -> l
   | l (feuillev(v))     -> insere v l
   | l (noeuds(_,fils))  -> aux l fils
   | l (noeudd(fg,_,fd)) -> aux (aux l fg) fd
in
   aux [] arbre;;
```

Pour vérifier qu'un arbre est une tautologie, si cet arbre ne contient aucune variable, on l'évalue sinon on remplace l'une des variable par `vrai` et `faux`. On obtient ainsi deux arbres logiques et l'on vérifie qu'ils représentent tous les deux une tautologie. Si un arbre logique contient n variables et que c'est une tautologie, on va construire ainsi 2^n arbres.

```
let rec remplace v c = function
    feuillec(a) as f -> f
    | feuillev(v') when v'=v -> feuillec(c)
    | feuillev(_) as f -> f
    | noeuds(fct,fils) -> noeuds(fct,remplace v c fils)
    | noeudd(fg,op,fd) -> noeudd(remplace v c fg,op,remplace v c fd);;

let rec tautologie a =
match liste_var a with
  [] -> evalexpr a
  | x::xs -> tautologie (remplace x vrai a) && (tautologie (remplace x faux a));;
```