TD 11 - Modélisation des formules propositionnelles

Exercice 1 [**Forme normale conjonctive et disjonctive**]

1. Donner les formes normales conjonctives et disjonctives de `xor`.
2. Caractériser la forme normale disjonctive d'une antilogie.
3. Donner les formes normales conjonctives et disjonctives de $a = \neg\big(x \wedge \neg(y \vee z)\big)$

Exercice 2 On veut pouvoir tester si une proposition logique est une tautologie. On définit pour cela le type récursif proposition :

```
type proposition =
  | Var of string
  | Et of proposition * proposition
  | Ou of proposition * proposition
  | Xor of proposition * proposition
  | Non of proposition
  | Implique of proposition * proposition
  | Equiv of proposition * proposition
;;
```

Par exemple la proposition $f := (P \wedge (P \to Q)) \to Q$ sera modélisée par

```
let f = Implique(Et(Var "P", Implique(Var "P", Var "Q")), Var "Q");;
```

1. On veut pouvoir récupérer les variables propositionnelles figurant dans une formule propositionnelle f, afin de construire une valuation et évaluer la proposition f sur cette valuation. Écrire une fonction `liste_variables` qui à une proposition associe la liste des variables propositionnelles qu'elle fait intervenir.
2. Écrire une fonction `print_prop` qui prend en entrée une proposition et qui renvoie cette proposition écrite de manière naturelle sous forme d'une chaîne de caractères.

```
# print_prop f;;
- : string = "((P)␣ET␣((P)␣=>␣(Q)))␣=>␣(Q)"
```

3. On représente valuation par un dictionnaire que l'on manipulera avec le module `Hashtbl` les clés étant les variables propositionnelles et les values associées sont données sous forme de booléens.
   a. Écrire une fonction `couple2dico : string list -> bool list -> (string, bool) Hashtbl.t` qui reçoit une liste de variables et une liste de valeurs et renvoie le dictionnaire précédemment définit.
   b. Écrire une fonction `evaluation : (string, bool) Hashtbl.t -> proposition -> bool` qui reçoit une valuation et une formule propositionnelle et renvoie l'évaluation de celle-ci.
   c. Écrire une fonction `affiche_valeurs : ('a, bool) Hashtbl.t -> 'a list -> unit` qui reçoit un dictionnaire et une liste de clés et affiche les valeurs correspondantes :

```
# let d = couple2dico ["a";"b";"c";"d"] [true ; true; false; false];;
val d : (string, bool) Hashtbl.t = <abstr>
# affiche_valeurs d ["a";"c";"d"];;
V       F       F       - : unit = ()
```

d. Ecrire une fonction `table : (string, bool) Hashtbl.t -> proposition -> unit` qui affiche la ligne de la table de vérité correspondant à la valuation donnée par le dictionnaire.

```
# let d = couple2dico ["P";"Q"] [true ; false];;
val d : (string, bool) Hashtbl.t = <abstr>
# table d f;;
P       Q       f
V       F       V
```

4. On cherche à présent à réaliser une table de vérité (complète). Pour cela il faut commencer par lister les 2^n possibilités (où n est le nombre de variables).

   a. Écrire une fonction `val colle : bool list list -> bool list list` qui reçoit une liste de listes et renvoie une liste de listes après avoir ajouté tour à tour `true` et `false` à chaque élément :

```
# colle [ [true] ; [false]];;
- : bool list list =
[[true; true]; [false; true]; [true; false]; [false; false]]
# colle [[]];;
- : bool list list = [[true]; [false]]
```

   b. En déduire une fonction `liste_valuations : int -> bool list list` qui recense tous les n-uplets de booléens possibles :

```
# liste_valuations 3;;
- : bool list list =
[[true; true; true]; [false; true; true]; [true; false; true];
 [false; false; true]; [true; true; false]; [false; true; false];
 [true; false; false]; [false; false; false]]
```

   c. Ecrire une fonction `table_verite : proposition -> unit` qui dresse la table de vérité d'une proposition :

```
# table_verite (Xor(Var "a",Var "b"));;
a       b       f
V       V       F
F       V       V
V       F       V
F       F       F
```

5. Écrire une fonction `est_tautologie` qui teste si une formule propositionnelle est une tautologie