TP 3 : Logique. Résolution naïve du problème SAT

On définit le type `log` (pour expression logique) comme suit :

```
type log=
  Un | Zero | V of string | Et of log * log | Ou of log * log | Non of log | Xor of log * log
  | Implique of log * log | Equivalent of log * log
;;
```

Remarquez que les variables sont des chaînes de caractères. On utilise tous les connecteurs vus en cours (excepté $\downarrow$ et $\uparrow$), ainsi que les constantes `Un` (1 ou Vrai) et `Zero` (0 ou Faux). L'interprétation arborescente vue en cours s'étend naturellement aux connecteurs que l'on a rajoutés ($\oplus$, $\Rightarrow$ et $\Leftrightarrow$).

Vous trouverez sur la page web le type et des exemples de formules comme :

```
let e  = Non (Xor (Et (V "a", V "b"), Et (V "a", V "c"))) ;;
```

1 Une petite intro

Question 1. *1.* On définit la hauteur d'une expression logique comme la hauteur de l'arbre binaire associé. Écrire une fonction `hauteur e` de signature `log -> int` calculant la hauteur d'une expression logique.

2. La longueur (taille) d'une expression logique et le nombre de nœuds de l'arbre associé. Écrire une fonction `longueur e` de signature `log -> int` calculant la hauteur de `e`.

```
# hauteur e, longueur e ;;
- : int * int = (3, 8)
```

2 Un testeur de tautologies

Question 2. Écrire une fonction `variables e` de signature `log -> string list` retournant la liste des variables apparaissant dans une expression logique. Attention : chaque variable apparaissant dans `e` doit apparaître une et une seule fois dans la liste résultat. On pourra introduire des fonctions auxiliaires, comme par exemple une fonction qui concatène deux listes en éliminant les doublons.

```
# variables e ;;
- : string list = ["c"; "b"; "a"]
```

On va maintenant écrire une fonction d'évaluation d'une expression logique. Outre l'expression cette fonction d'évaluation doit prendre également en entrée une distribution de vérité sur un ensemble de variables (contenant au moins toutes les variables de l'expression). Cette distribution de vérité est donnée sous la forme d'une liste de couples `(nom_de_variable, booléen)`, elle est donc de type `(string * bool) list`.

Question 3. Écrire une fonction `association x q` de type `'a -> ('a * 'b) list -> 'b` prenant en entrée un élément `x` de type `'a` est une liste `q` contenant des couples de la forme `'a * 'b` et retournant y tel que le couple (x, y) est le premier couple apparaissant dans la liste avec comme première composante x. (Remarque : `List.assoc` fait déja cela, mais il faut savoir la réécrire !)

Question 4. Écrire une fonction `evalue : log -> (string * bool) list -> bool` évaluant une expression logique avec une distribution de vérité.

```
# evalue e ["a", true; "b", false; "c",true] ;;
- : bool = false
```

Question 5. Écrire une fonction `distributions v` de type `string list -> (string * bool) list list` engendrant toutes les distributions de vérités sur `v`. On pourra utiliser `List.map`.

```
# distributions ["a"; "b"] ;;
- : (string * bool) list list = [[("a", true); ("b", true)]; [("a", true); ("b", false)];
[("a", false); ("b", true)]; [("a", false); ("b", false)]]
```

Question 6. En déduire une fonction `est_tautologie e` de signature `log -> bool`, retournant un booléen suivant si l'expression logique est une tautologie ou non.

```
# (est_tautologie f1, est_tautologie f2, est_tautologie f3) ;;
- : bool * bool * bool = (false, false, true)
```

Question 7. Parmi les formules $(((P \Rightarrow Q) \Rightarrow P) \Rightarrow P)$ et $(((P \Rightarrow Q) \Rightarrow P) \Rightarrow Q)$, laquelle est une tautologie ? (C'est la loi de Pierce).

3 Amélioration : affichage d'un contre-exemple

En utilisant une exception, il est possible d'améliorer la fonction précédente.

Question 8. Écrire une fonction `affiche d` de signature `(string * bool) list -> unit` affichant à l'écran une distribution de vérité. Par exemple :

```
#affiche_d  ["a", true; "b", false; "c", true] ;;
a --> 1      b --> 0      c --> 1
- : unit = ()
```

On utilisera les fonctions `print_int`, `print_string`, `print_newline`... Et on affichera 0 ou 1 plutôt que des booléens.

On déclare l'exception suivante, de type `(string * bool) list exn` :

```
exception Contre_ex of (string * bool) list ;;
```

Rappel : le mécanisme pour lever une exception est `raise` : par exemple ici avec `q` une `(string * bool) list`, on utiliserait `raise (Contre_ex q)`. Tant qu'elle n'est pas rattrapée, l'exception est propagée (on sort de toutes les fonctions appelées, jusqu'au programme principal, avec l'exception) Pour *rattraper* une exception, on encadre le code pouvant la produire avec `try` et `with` : on écrira `try code with Contre_ex q -> actions`. Ce mécanisme permet d'effectuer `actions` si l'exception `Contre_ex` est levée durant l'exécution de `code`. En fait on peut filtrer sur plusieurs exceptions, mais dans ce TP on n'aura pas besoin de cette possibilité.

Question 9. Écrire une fonction `est_tautologie e` semblable à la précédente, mais qui affiche un contre-exemple si la formule n'est pas une tautologie. On déclenchera une exception si on trouve une distribution de vérité qui ne satisfait pas l'expression logique, et on rattrapera l'expression pour afficher la distribution. On n'oubliera pas de renvoyer quand même `false` dans ce cas.

```
# est_tautologie f2 ;;
Ca ne marche pas avec la distribution:
b --> 1      a --> 0      c --> 1
- : bool = false
```

4 Résolution du problème SAT

Question 10. Reprendre les questions précédentes avec une fonction `est_satisfiable e` de signature `log -> bool`, affichant à l'écran un *certificat* si la formule est satisfiable (c'est à dire une distribution de vérité qui rend la formule vraie).

Remarque : la complexité de `est_tautologie` et `est_satisfiable` n'est pas « optimale » à cause de l'utilisation de listes pour encoder les distributions de vérité. Une utilisation d'une table de hachage mènerait à une complexité $O(2^n \ell)$ où ℓ est la longueur de la formule et n son nombre de variables (sous l'hypothèse de hachage uniforme simple !)