TP : Logique

D'après un travail de mon collègue M. Roux.
On considère le type

```
type expression =
| V
| F
| Var of string
| Non of expression
| Ou of expression * expression
| Et of expression * expression ;;
```

Prise en main

1. Écrire la fonction `ets (expression list) : expression` prenant en argument une liste d'expressions, et renvoyant le "et logique" de ces expressions (qu'on sait associatif).
   Idem pour `ous`
2. Écrire les fonctions `implique(e1,e2)` , `equivaut(e1,e2)` et `xor(e1,e2)` de types `expression * expression -> expression` qui renvoient respectivement des expressions équivalentes à $e_1 \implies e_2$, $e_1 \iff e_2$ $e_1 \oplus e_2$ (les fonctions prennent bien en argument un couple d'expressions, à l'instar des constructeurs).
3. Ecrire une fonction `substitution v f e` de type `string -> expression -> expression -> expression` qui remplace la variable `v` par l'expression `f` dans l'expression `e`.

Evaluation

Pour pouvoir évaluer une expression, on associe à chaque variable une valeur de vérité. Il est alors naturel de représenter une distribution de vérité grâce à une table d'association du type `(string * bool) list` (c'est- à-dire une liste de couples)

4. Ecrire une fonction `assoc a l` de type `'a -> ('a * 'b) list -> 'b` qui renvoie la valeur associée à la clé `a` dans la table d'association `l`. C'est-à-dire que `assoc a [ ...; (a,b); ...]` renvoie `b` si (a,b) est le couple le plus à gauche dont a est le premier terme. Et la fonction produit une erreur si aucune valeur n'est associée à a dans `l`.
5. En plusieurs étapes :
   — Écrire une fonction `evaluation e dv` de type `'a -> ('a * 'b) list -> 'b` qui renvoie l'évaluation de l'expression `e` en la distribution de vérité passée en argument.
   Par exemple, on pourra obtenir :

```
# let e3 = Ou ( Et (a,b) , Et (a,c) ) in
evaluation e3 [("a",true);("b",true);("c",true)];;
- : bool = true
```

— On note N le nombre de variables de e. Majorer N en fonction de $|e|$, la taille de e.
— Indiquer la complexité en fonction de la taille $|e|$ et N.

Sémantique

6. Écrire une fonction `cherche_variables e` de type `'a -> ('a * 'b) list -> 'b` qui renvoie la liste des variables (sans répétition) de l'expression `e` passée en argument. La fonction renvoie une liste de chaînes de caractères sans doublon.

```
# let a,b,c = Var "a", Var "b", Var "c" in
let e3 = Ou ( Et (a,b) , Et (a,c) ) in
cherche_variables e3;;
    - : string list = ["b"; "a"; "c"]
```

Définition 1. Le *backtracking* (retour sur trace) consiste à construire une solution à un problème petit à petit, en revenant en arrière s'il n'est pas possible de l'étendre en une solution complète

Remarque. Il s'agit en fait d'une exploration (exhaustive si on ne trouve pas de solution) de l'arbre de décision associé au problème. Exemples :
— Monnayeur en force brute.
— Recherche d'une sortie dans un labyrinthe.
— Résolution d'un sudoku en le remplissant case par case. S'il n'est pas possible de mettre un numéro dans une case, on revient en arrière sur le dernier choix effectué pour en choisir un autre.
— Coloriage d'un graphe avec k couleurs de façon à ce que 2 sommets adjacents soient de couleurs différentes
(Source : Quentin Fortier)

Pour déterminer si une expression est satisfiable, on affecte à chacune des variables les deux valeurs `true` et `false`. On opère donc une recherche par backtracking (on arrête l'exploration dès qu'on a trouvé un contexte qui satisfait la proposition).

8. Écrire une fonction `est_satisfiable` de type `expression -> bool` qui renvoie true ou false selon que l'expression passée en argument est ou non satisfiable. Quelle est la complexité de la fonction écrite, dans le pire cas et dans le meilleur cas ?

```
# let a,b,c = Var "a", Var "b", Var "c" in
let e3 = Ou ( Et (a,b) , Et (a,c) ) in
est_satisfiable e3;;
    - : bool = true

# let e = Et (Var "a", Non (Var "a")) in est_satisfiable e;;
- : bool = false
```

9. Écrire la fonction `antilogie e` de type `expression -> bool` qui détermine si e est une antilogie.

```
# let a,b,c = Var "a", Var "b", Var "c" in
let e3 = Ou ( Et (a,b) , Et (a,c) ) in
antilogie e3;;
    - : bool = false
# let e = Et (Var "a", Non (Var "a")) in antilogie e;;
- : bool = true
```

10. Écrire la fonction `tautologie e` de type `expression -> bool` qui détermine si e est une tautologie.

```
let e3 = Ou ( Et (a,b) , Et (a,c) ) in
tautologie e3;;
    - : bool = false
# let e = Ou (Var "a", Non (Var "a")) in tautologie e;;
- : bool = true
```

Algorithme de Quine

Dans cette section les formules sont en FNC. On définit

```
type litteral = V of int | NV of int
type cnf = litteral list list
```

1. Ecrire la fonction `litt_of_name_and_bool` de type `int -> bool -> litteral` qui prend en paramètre un nom de variable et un bouléen et construit le littéral correspondant positif si le bouléen est vrai et négatif sinon.

```
# litt_of_name_and_bool 2 true;;
- : litteral = V 2
# litt_of_name_and_bool 2 false;;
- : litteral = NV 2
```

2. Ecrire la fonction `subst` de type `litteral list list -> int -> bool -> litteral list list option` qui implémente la substitution décrite en cours.

```
# let f = [[V 1;NV 2; V 3];[V 1; V 2];[V 2; V 3];[V 3]];;
val f : litteral list list =
  [[V 1; NV 2; V 3]; [V 1; V 2]; [V 2; V 3]; [V 3]]
# subst f 2 true;;
- : litteral list list option = Some [[V 1; V 3]; [V 3]]
# subst f 3 false;;
- : litteral list list option = None
```

   Cette fonction substitue un bouléen à une variable connue par son nom (en clair, on entre en paramètre `3` et non `Var 3`)

3. Ecrire le fonction `get_var` de type `litteral list list -> int` qui retourne une variable de la proposition passée en argument (la première variable de la première clause)

```
# let f = [[V 1;NV 2; V 3];[V 1; V 2];[V 2; V 3];[V 3]];;
val f : litteral list list =
  [[V 1; NV 2; V 3]; [V 1; V 2]; [V 2; V 3]; [V 3]]
# get_var f;;
- : int = 1
```

```
# let f = [] in
get_var f;;
  Exception: Failure "pas de variable".
```

4. Ecrire la fonction `quine` qui implémente l'algorithme vu en cours.

```
# let f = [[V 1;NV 2; V 3];[V 1; V 2];[V 2; V 3];[V 3]];;
val f : litteral list list =
  [[V 1; NV 2; V 3]; [V 1; V 2]; [V 2; V 3]; [V 3]]
# quine f;;
- : bool = true
# quine [[V 1; NV 2];[NV 1; V 2];[NV 1; NV 2]];;
- : bool = true
# quine [[V 1; NV 2];[NV 1; V 2];[NV 1; NV 2]; [V 1;V 2]];;
- : bool = false
```