# *TD machine n°4*

**Exercice 1. Logique trivaluée**

On considère une logique trivaluée : `vrai, faux, indefini`.

1. En utilisant le type somme, définir un type `valeur_logique`.
2. On considère des formules logiques qui utilisent les trois valeurs du type `valeur_logique` et les opérateurs `et` et `non`, qui prennent respectivement deux et un argument(s). Écrire un type `formule_logique` dont les valeurs représentent des formules logiques. Par exemple :

```
let f = Et(Valeur Faux, Valeur Vrai) ;;
val f : formule_logique = Et(Valeur Faux, Valeur Vrai)
```

   `f` représente la formule logique `faux` et `vrai`.
3. Écrire une fonction `compte_et` qui calcule le nombre de `et` d'une formule.
4. Écrire une fonction `evaluer` qui calcule la valeur de vérité d'une formule. Sachant que

```
non indefini          =  indefini
indefini et vrai      =  indefini
indefini et faux      =  faux
indefini et indefini  =  indefini
```

   et que `et` est commutative. Exemples :

```
# let f1 = Et(Non(Valeur Faux), Valeur Vrai)
  and f2 = Et(Valeur Faux, Non(Valeur Vrai)) ;;
val f1 : formule_logique = Et(Non(Valeur Faux), Valeur Vrai)
val f2 : formule_logique = Et(Valeur Faux, Non(Valeur Vrai))

# let f3 = Et(Valeur Indefini, f2)
  and f4 = Et(f2, Valeur Indefini)
  and f5 = Et(Non f2, Valeur Indefini) ;;
val f3 : formule_logique =
    Et(Valeur Indefini, Et(Valeur Faux, Non(Valeur Vrai)))
val f4 : formule_logique =
    Et(Et(Valeur Faux, Non(Valeur Vrai)), Valeur Indefini)
val f5 : formule_logique =
    Et(Non(Et(Valeur Faux, Non(Valeur Vrai))), Valeur Indefini)

# let _ = List.map evaluer[f1; f2; f3; f4; f5];;
-: valeur_logique list = [Vrai; Faux; Faux; Faux; Indefini]
```

5. Écrire une fonction `equivalence_syntaxique` qui teste l'équivalence syntaxique de deux formules, c'est-à-dire qui renvoie `true` si les deux formules données en argument sont les mêmes. Exemple :

```
# equivalence_syntaxique f2 f3;;
- : bool = false
```

6. Écrire une fonction `equivalence_semantique` qui teste l'équivalence sémantique de deux formules, c'est-à-dire que les deux formules ont la *même valeur de vérité*. Exemple :

```
# equivalence_semantique f2 f3;;
- : bool = true
```

**Exercice 2.** On veut maintenant manipuler des formules de la logique trivaluée **avec variables.**

1. Donner un type polymorphe pour les formules avec variables, les variables étant représentées par des valeurs d'un type paramètre. La seule contrainte est que l'égalité doit être définie sur ces valeurs (les fonctions par exemple sont exclues).
2. Une formule qui n'a pas de variables est dite *formule close.* Écrire une fonction `est_close` qui indique si la formule logique avec variables est close ou non.
3. On appelle substitution l'opération qui consiste à remplacer toutes les occurrences d'une variable par une formule. Écrire une fonction `substitution` qui réalise une substitution.
4. Écrire une fonction `to_string` qui transforme une formule logique avec variables en chaîne de caractères, version lisible par un utilisateur de la formule.

**Exercice 3.** Le type `List` prédéfini dans Objective Caml permet d'avoir des éléments de même type dans une liste. Le but de cet exercice est de l'étendre au type `listIS` défini comme suit :

```
type listIS = Vide | I of int * listIS | S of string * listIS
```

**Exemple**

```
# let l = I(5,I(3,S("3",I(0,S("",Vide)))));;
val l : listIS = I (5, I (3, S ("3", I (0, S ("", Vide)))))
```

1. Écrire une fonction `afficheIS` qui permet d'afficher les éléments d'une liste donnée.
   ```
   # afficheIS l;;
   5 3 3 0
   - : unit = ()
   ```
2. Écrire une fonction `mapIS_liste` de type `(int -> 'a) -> (string -> 'a) -> listeIS -> 'a list`, qui permet de convertir une liste de type `listIS` en une liste classique.
   Voici un exemple d'application de cette fonction :
   ```
   # mapIS_liste (string_of_int ) (fun s -> s ) l;;
   - : string list = ["5"; "3"; "3"; "0"; ""]

   # mapIS_liste (fun x -> x = 0) (fun x -> x = "" ) l;;
   - : bool list = [false; false; false; true; true]
   ```
3. Écrire une fonction `liste_to_IS` qui permet de convertir une liste de type `list` en une liste de type `listeIS`. Pour écrire cette fonction on utilise la récupération d'exception en utilisant `try ... with`.
4. Écrire un type paramétré, récursif et polymorphe qui est la généralisation du type `listeIS`.

**Exercice 4. Dictionnaire** On considère la signature suivante :

```
type ('a, 'b) dictionnaire
val creer:unit -> ('a, 'b) dictionnaire
val ajouter: 'a -> 'b -> ('a, 'b) dictionnaire -> ('a, 'b) dictionnaire
val acceder: 'a -> ('a, 'b) dictionnaire -> 'b
val appartient: 'a -> ('a, 'b) dictionnaire -> bool
val supprimer: 'a -> ('a, 'b) dictionnaire -> ('a, 'b) dictionnaire
exception Pas_presente
```

Implanter un module `Dictionnaire` respectant l'interface ci-dessus et en prenant pour type concret :

```
type ('a, 'b) dictionnaire =
  Vide
| Element of 'a * 'b * ('a, 'b) dictionnaire ;;
```