TD de *Logique et Circuits* n° 8
(Correction)

Calcul Propositionnel

Exercice 1 On note P, M, J les propositions "Pierre est étudiant", "Marie est étudiante" et "Jean est étudiant". Traduire en formules de calcul propositionnel :

- Pierre et Marie sont étudiants
- Marie est étudiante ou Pierre n'est pas étudiant
- Pierre, Marie, et Jean ne sont pas tous les trois étudiants
- Ni Pierre ni Marie ne sont étudiants
- Si Pierre est étudiant, alors Marie et Jean sont étudiants
- Pierre est étudiant si Marie et Jean sont étudiants
- Marie est étudiante si et seulement si Pierre l'est

Correction : $P \wedge M$, $M \vee \neg P$, $\neg(P \wedge M \wedge J)$, $\neg P \wedge \neg M$, $P \rightarrow M \wedge J$, $M \wedge J \rightarrow P$, $(M \rightarrow P) \wedge (P \rightarrow M)$.

Exercice 2 Une contrée est peuplée de bons, qui disent toujours la vérité et de méchants, qui mentent toujours.

1. Un méchant peut-il dire "je suis un méchant" ?
2. A déclare : "Je suis méchant ou B est bon". Que sont A et B ?
3. A déclare : "Nous sommes tous méchants". B déclare : "Un seul de nous trois est bon". Que sont A, B et C ?
4. X se tient à l'embranchement de deux routes. Une va au paradis, l'autre en enfer. Quelle question doit-on poser à X pour savoir où est le paradis ?

Correction :

1. Non
2. A et B sont tous les deux bons
3. A et C méchants, B gentil
4. Poser P : "X est gentil" et Q : "la porte gauche va au paradis". On cherche f telle que la réponse de X à "f est-elle vraie?" est oui ssi Q est vraie. "Est-ce que $\neg P$ dirait Q?"

Exercice 3 Quelle est la formule correspondant à la fonction suivante :

a	b	c	f(a,b,c)
V	V	V	V
V	V	F	V
V	F	V	V
V	F	F	F
F	V	V	V
F	V	F	F
F	F	V	V
F	F	F	V

Correction : $c \vee (a \leftrightarrow b)$

Exercice 4

1. Définir en OCaml un type d'arbre correspondant aux formules logiques avec les connecteurs $\wedge, \vee, \neg, \rightarrow$.

   Correction :

```
type conn_bi = Et | Ou | Impl
type arbre = F of int | N of conn_bi * arbre * arbre | Non of arbre;;
```

2. Écrire en Ocaml une fonction de type `'a -> 'a list list -> 'a list list` qui ajoute un élément à chaque liste d'une liste. Par exemple,

```
# ajout 1 [[2; 3] ; [4; 5]];;
- : int list list = [[1; 2; 3]; [1; 4; 5]]
```

   Correction :

```
let ajout v = List.map (fun x -> v :: x);;
```

3. Écrire une fonction `combi : int -> bool list list` qui renvoie toutes les combinaisons possibles de n valeurs booléennes. Par exemple, pour $n = 3$, on doit retrouver toutes les combinaisons possibles de a,b,c de l'exercice précédent :

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

   Correction :

```
let rec combi n =
  match n with
    0 -> [[]]
  | _ -> (ajout true (combi (n - 1))) @ (ajout false (combi (n - 1)));;
```

4. On définit une fonction par une liste de booléens, correspondant aux valeurs que prend la fonction pour les n-uplets dans l'ordre fourni par `combi`. Par exemple, la fonction f de l'exercice précédent s'écrit :

```
val f : bool list = [true; true; true; false; true; false; true; true]
```

   Écrire en OCaml la fonction `liste_faux` qui prend en argument une liste de combinaisons (résultat de `combi`) et une fonction et qui renvoie les n-uplets des paramètres pour lesquels f prend la valeur $false$. Par exemple, pour la fonction f de l'exercice précédent :

```
# liste_faux (combi 3) f;;
- : bool list list = [[false; true; false]; [true; false; false]]
```

   Correction :
   Solution directe:

```
let rec liste_faux l1 l2 =
    match l1, l2 with
      [],[]   -> []
 | p::r,v::q -> let m = liste_faux r q in
                    if not v then p::m else m
 | _         -> failwith "erreur";;
```

   Solution plus abstraite:

```
let liste_faux l1 l2 =
```

```
List.fold_left2 (fun acc p v -> if not v then p :: acc else acc) [] l1 l2 ;;
```

```
List.fold_left2 f a [b1; ...; bn] [c1; ...; cn] est

 f (... (f (f a b1 c1) b2 c2) ...) bn cn
```

5. Question subsidiaire pour ceux qui finissent les feuilles de Td ! Écrire une fonction qui prend un nombre de variables et une fonction booléenne comme définie à la question 4 et qui renvoie l'arbre d'une formule correspondant à cette fonction. Par exemple,

```
# en_arbre 3 f;;
- : arbre =
N (Ou, N (Et, N (Et, Non (F 0), F 1), Non (F 2)),
 N (Et, N (Et, F 0, Non (F 1)), Non (F 2)))
```

Correction :
Solution directe:

```
let feuille b i = if b then F i else Non (F i);;

let rec convl l n = match l with
      []       -> failwith "aucune proposition"
    | hd::[]   -> feuille hd n
    | hd :: tl -> N(Et, feuille hd n, convl tl (n+1));;

let rec conv_ll ll = match ll with
     []       -> failwith "fonction valide"
   | hll::[]  -> convl hll 0
   | hll::tll -> N(Ou, convl hll 0, conv_ll tll);;

let en_arbre n f =
    let ll = liste_faux (combi n) f in conv_ll ll;;
```

Solution plus abstraite:

```
List.fold_left f a [b1; ...; bn] est f (... (f (f a b1) b2) ...) bn

let en_arbre n f =
  let l = liste_faux (combi n) f in
  let feuille b i = if b then F i else Non (F i) in
  let conv = function [] -> failwith "aucune proposition"
    | hd :: tl -> List.fold_left (
        fun (i, r) x -> i + 1, N (Et, r, feuille x i)) (1, feuille hd 0) tl in
  match l with
    [] -> failwith "fonction valide"
  | hd :: tl -> List.fold_left (
      fun acc x -> let sa = snd (conv x) in N (Ou, acc, sa)) (snd (conv hd)) tl
```