1 Représentation des formules booléennes

```
type binop = Et | Ou | Oubien | Impl | Equiv;;

type formule =
  | Vrai
  | Faux
  | Var  of string
  | Non  of formule
  | Bin  of binop * formule * formule
;;

let rec affiche par f = match f with
  | Vrai   -> print_string "Vrai"
  | Faux   -> print_string "Faux"
  | Var(x) -> print_string x
  | Non(g) -> print_string "Non("; affiche false g; print_string ")"
  | Bin(op,g,h) ->
      if par then print_string "(";
      affiche true g;
      print_string(match op with
                     | Et     -> " . "
                     | Ou     -> " + "
                     | Oubien -> " ++ "
                     |Impl    -> " => "
                     |Equiv   -> " <=> ");
      affiche true h;
      if par then print_string ")"
;;
let print_formule f = affiche false f; print_newline();;
```

2 Vérificateur de tautologie

1.

```
let rec evalue f = match f with
  | Vrai   -> true
  | Faux   -> false
  | Var x  -> failwith "la formule contient une inconnu"
  | Non(g) -> not(evalue g)
  | Bin(op,g,h) ->
      let a = evalue g and b = evalue h in
      match op with
        | Et     -> a && b
        | Ou     -> a || b
        | Oubien -> a <> b
        | Impl   -> (not a) or b
```

```
        | Equiv  -> a = b
;;
```

2.

```
let rec subs x y f = match f with
  | Var(v) -> if v = x then y else f
  | Non(g) -> Non(subs x y g)
  | Bin(op,g,h) -> Bin(op, subs x y g, subs x y h)
  | _ -> f
;;
```

3.

```
let rec tautologie f vars = match vars with
  | [] -> evalue(f)
  | x::suite -> tautologie (subs x Vrai f) suite
              & tautologie (subs x Faux f) suite
;;
```

4.
Il faut bien prendre garde à d'abord simplifier les opérandes d'une expression avant de simplifier l'expression elle-même, sans quoi la simplification effectuée est très limitée.

```
let rec simplifie f = match f with
  | Non(g) -> begin
      match (simplifie g) with
       | Vrai   -> Faux
       | Faux   -> Vrai
       | Non(h) -> h
       | h      -> Non h
      end
  | Bin(op,g,h) -> begin match op,simplifie(g),simplifie(h) with
      | Et, Vrai, b     -> b
      | Et, Faux, b     -> Faux
      | Et, a, Vrai     -> a
      | Et, a, Faux     -> Faux

      | Ou, Vrai, b     -> Vrai
      | Ou, Faux, b     -> b
      | Ou, a, Vrai     -> Vrai
      | Ou, a, Faux     -> a

      | Oubien, Faux, b -> b
      | Oubien, Vrai, b -> Non b
      | Oubien, a, Faux -> a
      | Oubien, a, Vrai -> Non a

      | Impl, Faux, b   -> Vrai
      | Impl, Vrai, b   -> b
```

```
      | Impl, a, Faux   -> Non a
      | Impl, a, Vrai   -> Vrai

      | Equiv, Vrai, b  -> b
      | Equiv, Faux, b  -> Non b
      | Equiv, a, Vrai  -> a
      | Equiv, a, Faux  -> Non a

      | _,g,h -> Bin(op,g,h)
    end
  | _ -> f
;;
```

5.

```
let rec tautologie2 f vars = match vars with
  | [] -> evalue(f)
  | x::suite -> tautologie (simplifie (subs x Vrai f)) suite
              & tautologie (simplifie (subs x Faux f)) suite
;;
```

En y réflechissant bien, cette nouvelle fonction n'est pas beaucoup plus efficace que la précédente car elle ne tire pas partie de l'éventuelle disparition d'une variable booléenne dans une formule simplifiée. Ce problème sera réglé avec la question 7.

6.

```
let rec tautologie3 f vars = match vars with
  | [] -> (evalue(f),[])
  | x::suite ->
      match tautologie3 (subs x Vrai f) suite with
        | (false,l) -> (false,(x^"=Vrai")::l)
        | (true,_) ->
        match tautologie3 (subs x Faux f) suite with
          | (false,l) -> (false,(x^"=Faux")::l)
          | (true,_)  -> (true,[])
;;
```

7.

Cette question est assez difficile. Elle demande de programmer plusieurs fonctions auxiliaires.

La première permet de *fusionner* deux liste de variables, sans générer de doublons. Elle requiert que les listes de départ soient triées pour l'ordre alphabétique.

```
let rec fusion = fun
  | [] l2             -> l2
  | l1 []             -> l1
  | (x1::q1) (x2::q2) -> if x1=x2 then fusion q1 (x2::q2)
                         else if x1<x2 then x1::(fusion q1 (x2::q2))
                                       else x2::(fusion (x1::q1) q2)
;;
```

```
#fusion ["a";"b";"c"] ["b";"d";"e";"f"];;
- : string list = ["a"; "b"; "c"; "d"; "e"; "f"]
```

La deuxième permet de calculer la liste des variables contenues dans une formule, en utilisant la fonction `fusion`.

```
let rec liste_vars f = match f with
  | Vrai   -> []
  | Faux   -> []
  | Var x  -> [x]
  | Non(g) -> liste_vars g
  | Bin(op,g,h) -> fusion (liste_vars g) (liste_vars h)
;;

#liste_vars (Bin (Et,Bin (Ou,Var "b",Var "c"),Bin (Impl,Var "b",Var "a")));;
- : string list = ["a"; "b"; "c"]
```

On en déduit une nouvelle version de `tautologie` :

```
let rec tautologie4 f = match (liste_vars f) with
  | [] -> evalue(f)
  | x::suite -> tautologie4 (simplifie (subs x Vrai f))
              & tautologie4 (simplifie (subs x Faux f))
;;
```

8.
Cette question demande beaucoup d'initiative.
On suppose que la formule à simplifier ne contient plus de constante `Vrai` ou `Faux`. On commence par exprimer les opérateurs `Oubien`, `Impl` et `Equiv` à l'aide de `Et`, `Ou` et `Non`.

```
let rec enleve_Oubien_Impl_Equiv f = match f with
  | Non(g) -> Non (enleve_Oubien_Impl_Equiv g)
  | Bin(op,g,h) ->
      begin
        let a = enleve_Oubien_Impl_Equiv g
        and b = enleve_Oubien_Impl_Equiv h in
        match op with
          | Et     -> Bin(Et,a,b)
          | Ou     -> Bin(Ou,a,b)
          | Oubien -> Bin(Ou,Bin (Et,Non a,b),Bin (Et,a,Non b))
          | Impl   -> Bin(Ou,Non a,b)
          | Equiv  -> Bin(Et,Bin (Ou,Non a,b),Bin (Ou,a,Non b))
      end
  | f -> f
;;

#let f=(Bin (Impl,Var "a",Bin(Oubien,Var "b",Var "c")));;
#let g=(enleve_Oubien_Impl_Equiv f);;
#print_formule g;;
```

```
Non(a) + ((Non(b) . c) + (b . Non(c)))
- : unit = ()
#print_formule f;;
a => (b ++ c)
- : unit = ()
#tautologie4 (Bin (Equiv,f,g)) ;;
- : bool = true
```

On peut alors faire la transformation demandée, en se basant sur les formules de distributivité suivantes

$$a + (b \cdot c) \equiv (a + b) \cdot (a + c) \quad \text{et} \quad (b \cdot c) + a \equiv (b + a) \cdot (c + a)$$

```
let rec transforme f = match f with
  | Non(Non g) -> transforme g
  | Non(Bin (Ou,g,h)) -> Bin (Et,transforme (Non g),transforme (Non h))
  | Non(Bin (Et,g,h)) -> transforme (Bin (Ou,Non g,Non h))
  | Bin(Et,g,h) -> Bin (Et,transforme g,transforme h)
  | Bin(Ou,g,h) -> begin
      match (transforme g) with
      | Bin(Et,a1,a2) -> Bin(Et,transforme (Bin(Ou,a1,h)),
                                transforme (Bin(Ou,a2,h)))
      | a -> begin
             match (transforme h) with
             | Bin(Et,b1,b2) -> Bin(Et,transforme (Bin(Ou,a,b1)),
                                       transforme (Bin(Ou,a,b2)))
             | b -> Bin(Ou,a,b)
             end
      end
  | f -> f
;;

#let h=(transforme g);;
#print_formule h;;
((Non(a) + (Non(b) + b)) . (Non(a) + (Non(b) + Non(c))))
   . ((Non(a) + (c + b)) . (Non(a) + (c + Non(c))))
- : unit = ()
#tautologie4(Bin(Equiv,h,f));;
- : bool = true
```

Une conjonction de formules $f_1, \ldots, f_n$ est une tautologie si et seulement si chaque f_i est une tautologie. Une disjonction de formules atomiques du type p ou $\overline{p}$, avec p une variable booléenne, est une tautologie si et seulement si il existe une variable booléenne q telle que, à la fois q et $\overline{q}$ apparaissent dans la disjonction. Grâce à ces remarques on obtient une nouvelle méthode pour vérifier les tautologies. Il reste cependant du travail à accomplir car il faut pouvoir détecter si une disjonction contient une variable booléenne et sa négation. Je vous laisse ce travail en exercice...