Manipulation de formules booléennes

On modélise les formules booléennes par des arbres selon la déclaration suivante :

```
type binop = Et | Ou | Oubien | Impl | Equiv;;
type formule =
  | Vrai | Faux
  | Var  of string
  | Non  of formule
  | Bin  of binop * formule * formule
;;
```

`Vrai` et `Faux` sont des formules constantes ; `Var(`x`)` représente une variable booléenne où x est la chaîne de caractères contenant le nom de cette variable ; `Non(`f`)` représente la formule $\overline{f}$; `Bin(Et,`f`,`g`)`, `Bin(Ou,`f`,`g`)`, `Bin(Oubien,`f`,`g`)`, `Bin(Impl,`f`,`g`)` et `Bin(Equiv,`f`,`g`)` représentent les formules fg, $f+g$, $f \oplus g$, $f \Longrightarrow g$ et $f \Longleftrightarrow g$. Par exemple si p et q sont deux variables booléennes, les formules :

$$f \equiv (p \Longrightarrow q) \Longleftrightarrow (\overline{q} \Longrightarrow \overline{p}) \qquad \text{et} \qquad g \equiv (p \Longrightarrow q) \Longrightarrow (q \Longrightarrow p)$$

sont représentées par :

```
    Bin(Equiv, Bin (Impl, Var "p", Var "q"),
           Bin (Impl, Non (Var "q"), Non (Var "p")))
```

et

```
    Bin (Impl, Bin (Impl, Var "p", Var "q"), Bin (Impl, Var "q", Var "p")).
```

Pour faciliter la saisie de telles formules on utilisera les opérateurs infixes `*,+,++,=>,<=>` définis par :

```
let prefix *   f g = Bin(Et,    f,g)
and prefix +   f g = Bin(Ou,    f,g)
and prefix ++  f g = Bin(Oubien,f,g)
and prefix =>  f g = Bin(Impl,  f,g)
and prefix <=> f g = Bin(Equiv, f,g)
;;
```

De même, pour améliorer l'affichage des formules manipulées, on utilisera la fonction suivante :

```
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();;
```

`print_formule` f affiche la formule f selon la notation infixe en plaçant des parenthèses autour des expressions composées quand c'est nécessaire. Exemple ;

```
 #let p = Var "p" and q = Var "q" and r = Var "r";;
p : formule = Var "p"
q : formule = Var "q"
r : formule = Var "r"
#let g = (p => q) => (q => p);;
g : formule =
 Bin (Impl, Bin (Impl, Var "p", Var "q"), Bin (Impl, Var "q", Var "p"))
#print_formule g;;
(p => q) => (q => p)
- : unit = ()
```

L'objectif de ce TP est d'étudier les manipulations élémentaires sur ce type de formules : évaluation d'une formule sans variables, substitution d'une formule à une variable, reconnaissance de l'identité de deux formules et simplification.

1) Évaluation d'une formule sans variables

On suppose que la formule f est constituée uniquement des constantes `Vrai`, `Faux` et des connecteurs `Non`, `Et`, `Ou`, `Oubien`, `Impl` et `Equiv`. Écrire une fonction `évalue : formule -> bool` qui retourne le booléen associé à cette expression. On devra obtenir par exemple :

```
 #evalue ((Vrai ++ Faux) => (Vrai => Faux));;
- : bool = false
#evalue ((Vrai => Faux) => (Vrai ++ Faux));;
- : bool = true
```

2) Substitution d'une formule à une variable

Écrire une fonction `subs : string -> formule -> formule` telle que `subs` x f g remplace dans la formule g toutes les occurrences de la variable x par la formule f. Par exemple :

```
 #print_formule(subs "p" (q+r) (p ++ Non(p)));;
(q + r) ++ Non(q + r)
- : unit = ()
```

3) Identité de deux formules

Si f et g sont deux formules dépendant des variables $p, q, r, \ldots$ on peut prouver l'identité de f et g en examinant tous les cas possibles pour $p, q, r, \ldots$ et en vérifiant dans chaque cas que les évaluations de f et g rendent la même valeur. En pratique on utilisera les règles suivantes :

- si f et g sont sans variables alors $f \equiv g$ si et seulement si `évalue`$(f) =$ `évalue`(g) ;
- si p est l'une des variables de f et g alors $f \equiv g$ si et seulement si :
  (`subs "p" Vrai` f) $\equiv$ (`subs "p" Vrai` g) et (`subs "p" Faux` f) $\equiv$ (`subs "p" Faux` g).

Programmer cela. On écrira une fonction : `identique : formule -> formule -> string list -> bool` qui dit si deux formules sont identiques. Le troisième argument est la liste des variables apparaissant dans l'une ou l'autre des deux formules à tester. Vérifier les identités usuelles : $p(q+r) \equiv pq+pr$, $\overline{p+q} \equiv \overline{p}\,\overline{q}$, $p \Longrightarrow q \equiv \overline{q} \Longrightarrow \overline{p}, \ldots$

Lorsque deux formules ne sont pas identiques il peut être intérressant de produire un cas où leurs évaluations diffèrent. Modifier la fonction précédente de sorte qu'elle retourne en cas de différence une liste de valeurs pour les variables contredisant l'identité. On définira un type spécial pour la valeur retournée :

```
type Resultat = Identique | Différent of bool list;;
```

4) Simplification

On peut simplifier *grossièrement* une formule f en évaluant les connecteurs dont un ou deux opérandes sont constants et en supprimant les doubles négations. Par exemple :

```
 #print_formule(simplifie ((p ++ Vrai) => Non(Non(q))));;
Non(p) => q
- : unit = ()
```