Formules logiques

On considère des formules propositionnelles définies par le type suivant :

```
type formule =
  | Const of bool
  | Var of string
  | Et of formule * formule
  | Ou of formule * formule
  | Non of formule
```

1 Affichage d'une formule logique

1. Écrire une fonction `string_of_formule` renvoyant la représentation infixe d'une expression sous la forme d'une chaîne de caractères. Le parenthésage doit être suffisant pour qu'il n'y ait pas d'ambiguïté (il peut être excessif).

   ```
   string_of_formule : formule -> string
   ```

   ```
   # string_of_formule (Ou (Et (Var "y", Const false), Non (Var "z")));;
   - : string = "((y et false) ou non z)"
   ```

2. Pour minimiser le nombre de parenthèses utilisées, on définit les priorités suivantes :
   — `Non` est prioritaire sur `Et` et `Ou` ;
   — `Et` est prioritaire sur `Ou`.
   Ainsi, `"non x_1 et x_2 ou non x_3"` signifie `"((non x_1) et x_2) ou (non x_3)"`.
   On en profitera également pour se débarrasser des parenthèses rendues inutiles par l'associativité des différents opérateurs : on écrira `"x_1 et x_2 et x_3"` plutôt que `"x_1 et (x_2 et x_3)` ou `(x_1 et x_2) et x_3"`.
   On donne la fonction `priorite : formule -> int` suivante :

   ```
   let priorite = function
     | Var _ | Const _ -> max_int
     | Ou _ -> 0
     | Et _ -> 1
     | Non _ -> 2
   ```

   Écrire une fonction `string_priorite` n'utilisant que les parenthèses nécessaires.

   ```
   string_priorite : formule -> string
   ```

   ```
   # string_priorite antinomie;;
   - : string = "non x_0 et (x_2 et x_0 ou x_1) et non x_1 et non x_2"
   ```

2 Égalité syntaxique modulo associativité et commutativité

Dans cette section, on considère qu'une formule logique est un arbre de type `formule`, mais que deux formules peuvent être syntaxiquement égales sans correspondre au même arbre. Plus précisément, deux formules seront dites syntaxiquement égales si l'on peut passer de l'une à l'autre par l'application répétée des règles suivantes :

— $A \wedge (B \wedge C) \simeq (A \wedge B) \wedge C$ (Asso-$\wedge$)
— $A \vee (B \vee C) \simeq (A \vee B) \vee C$ (Asso-$\vee$)
— $A \wedge B \simeq B \wedge A$ (Com-$\wedge$)
— $A \vee B \simeq B \vee A$ (Com-$\vee$)

2.1 Traitement artisanal de la commutativité

Dans cet exercice, on ne considère que les deux règles (Com-$\vee$) et (Com-$\wedge$).

1. Dans chacun des cas suivants, indiquer si les deux expressions sont égales modulo les transformations considérées :
   (a) $(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$ et $(x_2 \wedge x_3) \wedge (x_0 \wedge x_1)$;
   (b) $(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$ et $(x_2 \wedge x_3) \wedge (x_1 \wedge x_0)$;
   (c) $(x_0 \wedge x_1) \wedge (x_2 \wedge x_3)$ et $(x_2 \wedge x_0) \wedge (x_1 \wedge x_3)$.
2. Écrire une fonction `egal_com` décidant si deux formules sont égales modulo la commutativité.

```
egal_com : formule -> formule -> bool
```

   On devrait avoir :

```
# egal_commut ex2 ex3;;
- : bool = true
# egal_commut gros_ex1 gros_ex2;;
- : bool = false
```

2.2 Associativité

Gérer l'associativité, et plus encore la combinaison de la commutativité avec l'associativité, est plus délicat, surtout si l'on veut un temps de calcul raisonnable. Pour commencer, on ne peut plus se contenter d'arbres binaires. On décide donc de définir un nouveau type :

```
type formule_asso =
  | C of bool
  | V of int
  | EtA of formule_asso list
  | OuA of formule_asso list
  | N of formule_asso
```

L'idée est ensuite de définir un représentant canonique pour chaque classe d'équivalence modulo associativité et commutativité, et une manière efficace de calculer ce représentant.

En OCaml, tous les types à l'exception des types fonctionnels sont dotés d'une relation d'ordre (totale). Cette relation est définie sur les types de base (pour `bool`, on a `false < true`) et ensuite étendue aux types algébriques de la manière suivante :

— pour un type produit `t = a1 * a2 * ... * an`, on prend l'ordre lexicographique induit par les ordres préexistants sur `a1`, `a2`, ... `an` ;
— pour un type somme `t = A1 of t1 | ... | An of tn`, on a `A1 x1 < A2 x2 < ... < An xn` quels que soient `x1, ..., xn` (et bien sûr `Ai x < Ai y` ssi `x < y`).
Autrement dit, l'ordre induit sur un type somme est déterminé par l'ordre dans lequel les variantes apparaissent dans la définition du type.

Une expression de type `formule_asso` sera dite *canonique* si elle vérifie les conditions suivantes :
— aucun nœud `EtA` n'a d'enfant de la forme `EtA xs` ;
— aucun nœud `OuA` n'a d'enfant de la forme `OuA xs` ;
— pour chaque nœud de la forme `EtA enfants` ou `OuA enfants`, la liste `enfants` est triée par ordre croissant.

2.2.1 Fonctions préliminaires

1. Écrire une fonction `insere : 'a -> 'a list -> 'a list` insérant un élément dans une liste supposée triée.
2. Écrire une fonction `fusionne : 'a list -> 'a list -> 'a list` fusionnant deux listes supposées triées.

2.2.2 Égalité syntaxique

1. Écrire une fonction `canonique : formule -> formule asso` mettant une expression sous forme canonique.
2. Écrire une fonction `egal_syntaxe : formule -> formule -> bool` décidant l'égalité syntaxique (modulo associativité et commutativité) de deux expressions logiques.