TP 5 : Logique propositionnelle

Dans ce TP, les fonctions du module `List` sont librement utilisables (`List.mem`, `List.map` `List.exists`, `List.for_all`. . .)

Calcul propositionnel

On considère les formules de la logique propositionnelle construites avec des variables propositionnelles et les connecteurs $\neg$, $\wedge$ et $\vee$. Pour représenter ces formules en OCaml, on se servira du type suivant :

```
type formule =
  | Var of string
  | Neg of formule
  | Et of formule * formule
  | Ou of formule * formule
;;
```

Par exemple, la formule $p \wedge \neg(q \vee r)$ sera représentée par :

```
Et (Var "p", Neg(Ou(Var "q",Var "r")));;
```

1. Écrire une fonction

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

   qui, étant donnée une formule, renvoie une chaîne de caractères qui représente l'écriture infixe de cette formule, avec des parenthèses encadrant toutes les sous-formules de connecteur binaire.

   Par exemple, pour la formule ci-dessus, on obtiendra : `"(p Et Neg (q Ou r))"`.

2. (a) Écrire une fonction récursive

   ```
   reunion : 'a list -> 'a list -> 'a list
   ```

   qui effectue la réunion (sans répétition) de deux ensembles représentés par des listes.

   Par exemple,

   ```
   reunion ["a";"d";"c";"e"] ["b";"c"];;
   ```

   renvoie

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

   On pourra utiliser la commande `List.mem t l` *qui renvoie* `true` *si* `t` *appartient à la liste* `l`*, faux sinon.*

   (b) Écrire une fonction

   ```
   list_of_vars : formule -> string list
   ```

   qui, étant donné une formule, renvoie une liste sans répétitions de tous ses noms de variables.

   Par exemple,

   ```
   list_of_vars (Et (Ou(Var "p",Var "q"), Var "q"));;
   ```

   renverra

   ```
   -: string list = ["p"; "q"]
   ```

   à l'ordre des éléments près.

Évaluation des formules

Appelons *affectation* (ou *valuation*) toute liste d'association de type `(string * bool) list`. Une affectation permet de spécifier les valeurs de vérité des variables d'une formule. Chaque variable apparaît une fois au plus dans une affectation.
Par exemple, une valuation possible pour la formule `Et (Var "p", Var "q")` est `[("p", true), ("q", false)]`.
La valeur de vérité d'une formule f relativement à une valuation e est indéfinie, si les variables apparaissant dans f n'apparaissent pas toutes dans e. Sinon, cette valeur de vérité est obtenue en remplaçant dans f,
— les variables par les valeurs spécifiées par e,
— les connecteurs logiques par les opérations qui leur sont naturellement associées.

3. Écrire une fonction

   ```
   eval_formule : formule -> (string * bool) list -> bool
   ```

   calculant la valeur de vérité d'une formule relativement à une valuation donnée.
   Par exemple,

   ```
   eval_formule (Et (Var "p", Var "q"))[("p", true); ("q", false)] ;;
   ```

   donnera

   ```
   -: bool = false
   ```

   Vous pouvez utiliser `List.assoc v l` qui renvoie la partie droite `b` du premier couple de la liste `l` de la forme `(v, b)` s'il existe, et déclenche une exception si ce couple est introuvable ; ou bien vous pouvez définir vous-même une fonction `association v l`.

Formules satisfiables et tautologies

Une formule de la logique propositionnelle est appelée une tautologie, si elle s'évalue à `true` pour tout affectation spécifiant les valeurs de vérité de toutes ses variables. Une formule qui s'évalue à `true` pour au moins une affectation est dite satisfiable.

4. Écrire une fonction polymorphe

   ```
   add_to_all : 'a -> 'a list list -> 'a list list
   ```

   telle que

   ```
   add_to_all x [l1; ... ; ln];;
   ```

   renvoie la liste de listes

   ```
   -: 'a list list = [x::l1; ... ; x::ln]
   ```

   On rappelle que, étant données une fonction `f` et une liste `l=[a1;...;an]`, la commande `List.map f l` renvoie la liste `[f(a1); ...; f(an)]`.
   On pourra au choix : soit utiliser la commande `List.map`, soit utiliser une fonction récursive auxiliaire.

5. En vous servant de la fonction `add_to_all`, écrire une fonction

   ```
   affectations_vars : string list -> (string * bool) list list
   ```

qui, étant donnée une liste de noms de variables, construit la liste de tous les affectations possibles (valuations) associées.

Par exemple,

```
affectations_vars ["q"];;
```

donnera

```
-: (string * bool) list list = [[("q", false)]; [("q", true) ]]
```

et

```
affectations_vars ["p"; "q"];;
```

donnera

```
-: (string * bool) list list = [[("p", false); ("q", false)];
[("p", false); ("q", true) ]; [("p", true); ("q", false)];
[("p", true); ("q", true) ]]
```

En déduire une fonction

```
affectations : formule -> (string * bool) list list
```

qui étant donnée une formule donne une liste de tous les valuations associées à ses variables.

6. Écrire une fonction

```
satisfiable : formule -> bool
```

qui, étant donnée une formule, détermine si elle est satisfiable, et une fonction

```
tautologie : formule -> bool
```

qui, étant donnée une formule, détermine si elle est une tautologie. Par exemple,

```
satisfiable (Et (Var "p", Neg(Var "p")));;
```

donnera

```
-: bool = false
```

et

```
tautologie (Ou(Var "p", Neg(Var "p")));;
```

donnera

```
-: bool = true
```

On rappelle que, étant donnée une fonction `f : 'a -> bool` *et une liste* `l : 'a list`, `List.exists f l` *renvoie* `true` *si et seulement si il existe* $x \in l$ *tel que* $f(x)$ *est vrai.*

Et que `List.for_all f l` *renvoie* `true` *si et seulement si pour tout* $x \in l$, $f(x)$ *est vrai.*

Par exemple, `List.exists (function x -> (x>6)) [7;8;9;6];;` *renvoie* `true` *et* `List.for_all (function x -> (x>6)) [7;8;9;6];;` *renvoie* `false`.

On pourra s'aider de ces fonctions ou bien de fonctions récursives auxiliaires.

Conséquences logiques et équivalences sémantiques

Si f et g sont deux formules de la logique propositionnelle, on dit que g est conséquence logique de f si, pour toute affectation pour laquelle f s'évalue à `true`, g s'évalue aussi à `true`.
Les affectations considérées doivent spécifier à la fois les valeurs des variables de f et de celles de g. Si g est conséquence logique de f et f est conséquence logique de g, on dit que ces deux formules sont sémantiquement équivalentes.

7. Écrire une fonction

   ```
   consequence : formule -> formule -> bool
   ```

   qui, étant données une formule f et une formule g, détermine si g est conséquence logique de f.

   En déduire une fonction

   ```
   equivalentes : formule -> formule -> bool
   ```

   qui détermine si ses deux arguments sont des formules sémantiquement équivalentes.

Forme normale conjonctive

Dans la logique propositionnelle les propriétés suivantes sont vérifiées :

- Descente des négations vers les variables. Les formules suivantes sont sémantiquement équivalentes :
  — $\neg(\neg F)$ et F
  — $\neg(F \vee G)$ et $(\neg F \wedge \neg G)$,
  — $\neg(F \wedge G)$ et $(\neg F \vee \neg G)$.
- Descente des $\vee$ sous les $\wedge$ par factorisation. Les formules suivantes sont sémantiquement équivalentes :
  — $(F \vee (G \wedge H))$ et $(F \vee G) \wedge (F \vee H)$,
  — $((F \wedge G) \vee H)$ et $(F \vee H) \wedge (G \vee H)$.

Une formule sera dite sous forme normale conjonctive si elle est, au parenthésage et à permutation près, de la forme $F_1 \wedge ... \wedge F_n$ où chaque F_i est de la forme $(\neg a_1 \vee ... \vee \neg a_m \vee b_1 \vee ... \vee b_p)$, où les a_i et b_j sont des variables propositionnelles.

8. A l'aide des propriétés ci-dessus, écrire les fonctions suivantes. Chacune, à partir d'une formule, doit produire une formule équivalente de la forme spécifiée.
   — `desc_neg : formule -> formule`. Cette fonction doit faire descendre les négations d'une formule vers ses variables, en simplifiant les négations doubles et en appliquant les lois de De Morgan.
   — `desc_ou : formule -> formule`. Cette fonction doit faire descendre les $\vee$ sous les $\wedge$ dans toute formule dans laquelle les négations ne s'appliquent qu'à des variables.
   — `fnc : formule -> formule`. Cette fonction doit transformer toute formule en sa forme normale conjonctive.