TP n°4

Logique propositionnelle en OCAML

On considère les formules de la logique propositionnelle construites à partir des variables propositionnelles et les connecteurs $\neg$, $\wedge$ et $\vee$.

Pour représenter une formule en Caml, on se servira du type suivant :

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

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

```
let f = Et (Vrai, Neg(Ou(Var "p",Var "r")));;
```

Exercice 1. Écrire une fonction

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

qui, étant donnée une formule, renvoie une chaîne de caractères qui représente cette formule écrite avec parenthèses. Par exemple, pour la formule `f` ci-dessus on obtiendra `"(Vrai Et Neg((p Ou r)))"`.

Exercice 2. Écrire une fonction

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

qui, étant donné une formule, renvoie une liste de toutes ses variables (sans élement dupliqué) dans l'ordre alphabétique. Par exemple, `list_of_vars (Et (Ou(Var "p",Var "q"), Var "q"))` renvoie `["p"; "q"]`

Exercice 3. Appelons *environnement* toute liste d'association de type `(string * bool) list`. Un environnement permet de spécifier les valeurs des variables d'une expression. Chaque variable peut apparaître une seule fois dans l'environnement. Par exemple, pour la formule `f` ci-dessus un environnement est `[("p",true),("r",false)]`

La *valeur d'une expression* `e` *dans l'environnement* `l` est soit indéfinie, si toutes les variables apparraissant dans `e` ne sont pas définies, sinon la valeur booléenne obtenue en remplaçant dans `e` les variables par les valeurs spécifiées par `l`, ainsi que les constantes par leur valeur booléene et tous les connecteurs logiques par les opérations qui leur sont naturellement associées. Écrire une fonction

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

qui évalue une formule étant donnée un environnement.
Par exemple, `eval_formule f [("r",true);("p",false)]` donnera `false`.

Vous pouvez utiliser `List.assoc v l` qui renvoie la partie droite `f` du premier couple de `l` de la forme `(v, f)` s'il existe, et déclenche une exception si ce couple est introuvable.

Simplification de formule

Une formule peut ne contenir aucune variable – elle est alors appelée *formule constante*. La valeur d'une telle formule est directement calculable, sans environnement. Même lorsque une formule contient des variables, certaines parties de cette formule peuvent être constantes : dans ce cas, la formule peut être simplifiée en remplaçant ces sous-formules par leurs valeurs.

Exercice 4. Écrire une fonction

```
eval_sous_formule : formule -> formule
```

telle que `eval_sous_formule f` renvoie la formule obtenue en remplaçant chaque sous-formule constante de `f` par sa valeur. On notera que si `f` est elle-même une formule constante, la valeur renvoyée doit être la formule `f`. D'autre part, si `e1` et `e2` sont constantes, alors `Neg(e1)`, `Et(e1,e2)` et `Or(e1,e2)` sont aussi des expressions constantes, et peuvent être remplacées par leur valeur. Par exemple, `eval_sous_formule (Et (Var "p", Ou(Faux,Faux)))` donne `Et (Var "p", Faux)`.

Exercice 5. Même lorsqu'elles sont non constantes, certaines parties d'une formule peuvent malgré tout être simplifiées au moyen des règles suivantes. Etant donnée une sous-expression d'une expression quelconque, on peut la remplacer :
- par `Vrai`, si elle est de la forme `Ou(Vrai,f)` ou `Ou(f,Vrai)`,
- par `Faux`, si elle est de la forme `Et(Faux,f)` ou `Et(f,Faux)`,

Écrire une fonction

```
simplifie_formule : formule -> formule
```

qui implémente les règles ci-dessous.

Par exemple, `simplifie_formule (Et (Var "p", Ou(Faux,Faux)))` donne `Faux`.
($\star$) Donner d'autres règles de simplification et réaliser-les. Par exemple, $A \wedge A$ est équivalent à A.

Forme normale conjonctive

Dans la logique propositionnelle les propriétés suivantes sont vérifiés :

1. *Descente des négations vers les variables - Doubles négations et lois de De Morgan.* Les formules suivantes sont équivalentes :
   - $\neg\neg F$ et F
   - $\neg(F \vee G)$ et $(\neg F \wedge \neg G)$,
   - $\neg(F \wedge G)$ et $(\neg F \vee \neg G)$.
2. *Descente des $\vee$ sous les $\wedge$ par factorisation.* Les formules suivantes sont équivalentes :
   - $(F \vee (G \wedge H))$ et $(F \vee G) \wedge (F \vee H)$,
   - $((F \wedge G) \vee H)$ et $(F \vee H) \wedge (G \vee H)$.

Exercice 6. Écrire une fonction

```
fnc : formule -> formule
```

qui transforme une formule vers une formule équivalente en forme normale conjonctive en utilisant les propriétés ci-dessus.

Par exemple, `fnc (Ou (Neg (Et (Var "p", Var "q")), Et (Var "q", Var "r")))` donne :

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

Les tautologies

Une formule de la logique propositionnelle est appelée une *tautologie*, si elle s'évalue à `true` pour tout environnement qui donne des valeurs booléennes à toutes ses variables.

Exercice 7. Écrire une fonction

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

qui étant donnée une formule donne une liste de tous ses environnements (une valeur booléenne pour chaque variable de la formule). Par exemple,
`environnements (formule = Et (Vrai, Neg (Ou (Var "p", Var "r")))` donne

```
[[("r", false); ("p", false)]; [("r", false); ("p", true)];
 [("r", true); ("p", false)]; [("r", true); ("p", true)]]
```

Exercice 8. Écrire une fonction

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

qui, étant donnée une formule, détermine si elle est une tautologie ou pas. Par exemple,
`tautologie (Ou(Var "p",Neg(Var "p")))` donne `true`.

L'opération XOR

Exercice 9. ($\star$) Ajouter l'opération XOR aux formules et refaire tous les exercices.