TP 4 : Déduction naturelle, algorithme de Quine

I. TD : Déduction naturelle

Donner une dérivation en déduction naturelle de chacun des séquents ci-dessous :

1. $\neg(p \vee q) \vdash \neg p \wedge \neg q$
2. $p \vee q \vdash q \vee p$
3. $p \to q \vdash (p \wedge r) \to (q \wedge r)$
4. $\vdash (p \to q) \to \neg(p \wedge \neg q)$
5. $p \to q, p \to \neg q \vdash \neg p$
6. $\neg p \vdash p \to q$
7. $\vdash \neg(p \wedge \neg p)$
8. $\neg\neg p, p \vee \neg p \vdash p$
9. $\vdash \neg\neg\neg p \to \neg p$

II. TP : Algorithme de Quine

Dans cet exercice, comme dans le TP précédent, on représente des formules logiques en OCaml avec le type suivant :

```
type formule =
| Vrai
| Faux
| Var of string
| Non of formule
| Et of formule * formule
| Ou of formule * formule
| Impl of formule * formule
```

Comme dans le TP précédent, une valuation est représentée par une liste de type `(string * bool) list`, contenant des couples de nom de variable et de valeur booléenne.

L'algorithme de Quine a pour but de trouver une valuation qui satisfait la formule. Il repose sur des simplifications de formules. Si on substitue une variable propositionnelle par une valeur `Vrai` ou `Faux`, on peut en déduire des simplifications par les équivalences de formules logiques.

II. A. Simplification de formules

Simplifier les éléments de base `Vrai`, `Faux` ou `Var` ne peuvent pas être simplifiés. En revanche, à l'aide des règles de simplification vues en cours, on peut créer des fonctions `simpl_non`, `simpl_et`, `simpl_ou` et `simpl_impl` qui simplifient les formules auxquelles elles s'appliquent lorsque c'est possible. Ces fonctions sont à définir par vos soins. Après cela, on pourra définir la fonction `simplifie : formule → formule` suivante :

```
let rec simplifie f =
  match f with
  | Vrai | Faux | Var _ → f (* aucune simplification possible *)
  | Non f'        → simpl_non  (simplifie f')
  | Et (f1, f2)   → simpl_et   (simplifie f1) (simplifie f2)
  | Ou (f1, f2)   → simpl_ou   (simplifie f1) (simplifie f2)
  | Impl (f1, f2) → simpl_impl (simplifie f1) (simplifie f2)
```

1. Écrire la fonction `simpl_non : formule → formule` qui construit la négation logique de la formule passée en paramètre, en simplifiant si c'est possible. Par exemple :
   - si la formule passée en paramètre est la variable x, on renvoie la formule $\neg x$.
   - si la formule est $\neg\varphi$ avec φ une formule quelconque, on peut renvoyer simplement φ.
   - si la formule est `Vrai`, on renvoie `Faux` et vice-versa.
2. Écrire la fonction `simpl_et : formule → formule → formule` qui construit la conjonction des formules passées en paramètre, en simplifiant si c'est possible.
3. Écrire la fonction `simpl_ou : formule → formule → formule` qui construit la disjonction des formules passées en paramètre, en simplifiant si c'est possible.
4. Écrire la fonction `simpl_impl : formule → formule → formule` qui construit l'implication des formules passées en paramètre, en simplifiant si c'est possible.

II. B. Implémentation de l'algorithme

Pour φ et ψ deux formules et x une variable propositionnelle, on note $\varphi\{x \leftarrow \psi\}$ la *substitution de la variable x par la formule ψ dans la formule φ*. Cette substitution consiste à remplacer chaque occurence de x par ψ dans φ. Par exemple, si $\varphi = p \vee q \rightarrow p$ et $\psi = x \wedge y$, alors $\varphi\{p \leftarrow \psi\} = (x \wedge y) \vee q \rightarrow (x \wedge y)$

On donne alors l'algorithme de Quine en pseudo-code :

```
1:  fonction Quine(φ) :
2:      soit φ' = simplifie(φ)
3:      si φ' = ⊤ alors :
4:          renvoyer Vrai
5:      sinon si φ' =⊥ alors :
6:          renvoyer Faux
7:      sinon :
8:          soit x = une variable de φ'
9:          renvoyer Quine(φ'{x ← ⊤})  ou  Quine(φ'{x ←⊥})
```

5. Écrire une fonction **`substitution : formule → string → formule → formule`**, telle que **`substitution f1 "x" f2`** renvoie une nouvelle formule, dans laquelle chaque occurence de la variable **`Var "x"`** dans **`f1`** a été remplacée par **`f2`**.
6. Écrire une fonction de **`quine_sat : formule → bool`**, qui implémente l'algorithme de quine. La fonction prend une formule en paramètre, et renvoie vrai si la formule est satisfiable, faux sinon.
7. Tester la fonction sur la formule suivante : $(x \wedge \neg z) \rightarrow (\neg x \wedge \neg(y \wedge \neg z))$
8. Adapter la fonction **`quine_sat`** pour qu'elle renvoie une valuation rendant la formule vraie si celle-ci est satisfiable.