Correction TP 5 : Logique propositionnelle

1.
```
type formule = Var of string
             | Neg of formule
             | Et of formule * formule
             | Ou of formule * formule;;

let f_ex=Et (Var "p", Neg(Ou(Var "q",Var "r")));;

let rec string_of_formule f = match f with
    Var s -> s
  | Neg f -> "Neg"^(string_of_formule f)
  | Et(f1,f2) -> "("^(string_of_formule f1)^" Et "^(string_of_formule f2)^")"
  | Ou(f1,f2) -> "("^(string_of_formule f1)^" Ou "^(string_of_formule f2)^")"
;;

string_of_formule f_ex;;
```

2.
```
let rec reunion l1 l2 = match l1 with
  |[] -> l2
  |t::q -> match List.mem t l2 with
    |true -> reunion q l2
    |false -> t::(reunion q l2);;

let rec list_of_vars f = match f with
    Var s -> [s]
  | Neg f -> list_of_vars f
  | Et(f1,f2)
  | Ou(f1,f2) ->  reunion (list_of_vars f1) (list_of_vars f2)
;;

list_of_vars f_ex;;
```

3.
```
let rec association v l=match l with
  |[] -> failwith "liste vide"
  |(a,b)::q when (a=v)-> b
  |t::q -> association v q;;

let rec eval_formule f l = match f with
    Var s -> (association s l)
  | Neg f1 -> not (eval_formule f1 l)
  | Et(f1,f2) -> (eval_formule f1 l) && (eval_formule f2 l)
  | Ou(f1,f2) -> (eval_formule f1 l) || (eval_formule f2 l)
;;

eval_formule f_ex ["p", true; "q", false; "r", true];;
```

4.
```
let add_to_all x ll = List.map (fun l -> x::l) ll
;;
```

5.
```
let rec affectations_vars l = match l with
  |[] -> [[]]
  | t::q ->
      let le = affectations_vars q in
      (add_to_all (t, false) le)@
      (add_to_all (t, true) le)
;;

affectations_vars ["a";"b"];;

let affectations f =
  affectations_vars (list_of_vars f)
;;

affectations f_ex;;
```

6. On pouvait proposer les fonctions :

```
let satisfiable f =
  let rec aux l= match l with
    |[] -> false
    |t::q -> (eval_formule f t)||(aux q) in
  aux (affectations f);;

let tautologie f =
  let rec aux l= match l with
    |[] -> true
    |t::q -> (eval_formule f t)&&(aux q) in
  aux (affectations f);;

satisfiable f_ex;;

tautologie f_ex;;
```

Ou bien, on pouvait aussi proposer les fonctions :

```
let satisfiable f =
  let le = affectations f in
  List.exists (eval_formule f) le;;

let tautologie f =
  let le = affectations f in
  List.for_all (eval_formule f) le;;
```

7. On povait proposer :

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

let consequence f g =
  let vf = list_of_vars f
  and vg = list_of_vars g in
```

```
  let v = reunion vf vg in
  let affs = affectations_vars v in
  let rec aux l= match l with
    |[] -> true
    |t::q -> ((not (eval_formule f t))|| (eval_formule g t))&&(aux q) in
  aux affs;;

let equivalentes f g =(consequence f g) && (consequence g f);;

consequence f_ex g_ex;;

equivalentes f_ex g_ex;;
```

On pouvait aussi proposer :

```
let consequence f g =
  let vf = list_of_vars f
  and vg = list_of_vars g in
  let v = reunion vf vg in
  let affs = affectations_vars v in
  List.for_all (fun e -> not (eval_formule f e) || (eval_formule g e) ) affs
;;
```

8.
```
let rec desc_neg f = match f with
    Neg(Neg g) -> desc_neg g
  | Neg(Ou(g,h)) -> Et(desc_neg(Neg g), desc_neg(Neg h))
  | Neg(Et(g,h)) -> Ou(desc_neg(Neg g), desc_neg(Neg h))
  | Ou(g,h) -> Ou(desc_neg g, desc_neg h)
  | Et(g,h) -> Et(desc_neg g, desc_neg h)
  | f -> f
;;


let rec desc_ou f = match f with
  | Et(g,h) -> Et(desc_ou g, desc_ou h)
  | Ou(g,h) ->
      let g'=desc_ou g
      and h'=desc_ou h in
      (match g',h' with
       | _,Et(a,b) -> Et(Ou(g',a), Ou(g',b))
       | Et(a,b),_ -> Et(Ou(a,h'),Ou(b,h'))
       | _ -> Ou(g',h'))
  | f -> f
;;

desc_neg f_ex;;

let fnc f = desc_ou (desc_neg f);;
```

```
f_ex;;
fnc f_ex;;

fnc g_ex;;

let p_ex=Neg(Et(Var("p"),Ou(Var("q"),Var("r"))));;

fnc p_ex;;
```