```
(***************************************************************************)
(*                          Methode de resolution                          *)
(***************************************************************************)

(* Le type des propositions : *)

type proposition_atomique == string;;

type proposition =
    Atome of proposition_atomique
  | Vrai
  | Faux
  | Neg of proposition
  | Imp of proposition * proposition
  | Et  of proposition * proposition
  | Ou  of proposition * proposition
;;

(* Exemples : *)

let A = Atome "A" and B = Atome "B" in Imp (A, Imp (Imp (A,B), B));;

(* Clauses et formes clausales *)

type clause == proposition list;;
type forme_clausale == clause list;;


(* union : 'a list -> 'a list -> 'a list *)

let union l1 l2 =
  let rec add x = function
    []          -> [x]
  | (a::r) as l -> if a=x then l else a::(add x r)
  in
  let rec iter acc = function
    []   -> acc
  | a::r -> iter (add a acc) r
  in
  iter (iter [] l1) l2
;;

(* complexite : n(n-1)/2,  ou n = | l1 | + | l2 | *)


(* prod : forme_clausale -> forme_clausale -> forme_clausale *)

let (prod : forme_clausale -> forme_clausale -> forme_clausale) = fun a b ->
  let rec prod_aux ai = function
    []     -> []
  | bj::bl -> (union ai bj) :: (prod_aux ai bl)
  in
  let rec iter = function
    []     -> []
  | ai::al -> (prod_aux ai b) @ (iter al)
  in
  iter a
;;

(* Complexite : somme(i) somme(j) (ai+bj)(ai+bj-1)/2
 *
 *            < n x m x max(ai+bj)^2 / 2
 *
```



```
 * parce que l'on peut se passer du @ en utilisant un accumulateur. *)


(* Forme clausale :
 *
 * Terminaison : la taille de la proposition pour laquelle on definit C(P)
 * decroit strictement, et tous les cas sont etudies. On peut prendre comme
 * taille :
 *              t(Atome _) = 1
 *              t(Faux) = t(Vrai) = 1
 *              t(A\/B) = t(A/\B) = t(A->B) = 1 + t(A) + t(B)
 *              t(~A) = 1 + t(A)                                           *)

(* fnc : proposition -> forme_clausale *)

let rec (fnc : proposition -> forme_clausale) = fun p ->
  match p with
    Atome _ | Neg (Atome _) | Faux | Neg Faux
                    -> [[p]]
  | Vrai            -> [[Neg Faux]]
  | Neg Vrai        -> [[Faux]]
  | Et (a,b)        -> let ca = fnc a and cb = fnc b in union ca cb
  | Ou (a,b)        -> let ca = fnc a and cb = fnc b in prod ca cb
  | Imp (a,b)       -> let ca = fnc (Neg a) and cb = fnc b in prod ca cb
  | Neg (Et (a,b))  -> let ca = fnc (Neg a) and cb = fnc (Neg b) in prod ca cb
  | Neg (Ou (a,b))  -> let ca = fnc (Neg a) and cb = fnc (Neg b) in union ca cb
  | Neg (Imp (a,b)) -> let ca = fnc a and cb = fnc (Neg b) in union ca cb
  | Neg (Neg a)     -> fnc a
;;

(* Complexite :
 *
 * Si h est la hauteur de la proposition P alors C(P) contient au plus
 * 2^(2^h) clauses.
 * Chaque clause contient des literaux distincts et en contient donc au
 * maximum 2 x (2^h).
 *)


(* Applications : *)

let A = Atome "A" and B = Atome "B";;

let F1 = Imp (Et (A,B) , A);; fnc F1;;
  (* [[Neg (Atome "A"); Neg (Atome "B"); Atome "A"]] *)

let F2 = Imp (A , Et (A,B));; fnc F2;;
  (* [[Neg (Atome "A"); Atome "A"]; [Neg (Atome "A"); Atome "B"]] *)

let F3 = Imp (A , Imp (Imp (A,B) , B));; fnc F3;;
  (* [[Neg (Atome "A"); Atome "A"; Atome "B"];
      [Neg (Atome "A"); Neg (Atome "B"); Atome "B"]] *)


(* enleve : 'a list -> 'a -> 'a list *)

let enleve l x =
  let rec enlev = function
    []   -> []
  | a::r -> if a=x then enlev r else a::(enlev r)
  in
  enlev l
;;
```

```
(* resolutions : clause -> clause -> clause list *)

let resolutions p q =
  let parcours l1 l2 =
    let rec parcours_rec = function
        []          -> []
      | (Neg a)::l -> if mem a l1 then
                        let r = union (enleve l1 a) (enleve l2 (Neg a)) in
                        (if r=[] then [Faux] else r) :: (parcours_rec l)
                      else
                        parcours_rec l
      | _::l       -> parcours_rec l in
    parcours_rec l2 in

  (parcours p q) @ (parcours q p)
;;

(* Complexite :
 *
 * Pour deux clauses de taille <= n il ne peut y avoir plus de n resolutions,
 * car on suppose les literaux des clauses distincts, ce qui sera vrai.
 * Chaque resolution est une clause de taille <= n-1.
 *)


(* succes : clause list -> bool *)

let succes l =
  mem [Faux] l
;;


(* decision : proposition -> bool *)

(* On commence par definir la fonction paires qui, pour une liste
 * l=[n1;n2;...;np], renvoie la liste de tous les {ni,nj},l\{ni,nj}. *)

let paires l =
  let rec prod_rec x = function
        []   -> []
      | y::r -> let l' = enleve (enleve l x) y in
                ((x,y) , l') :: (prod_rec x r)
  in
  let rec iter = function
        []   -> []
      | x::r -> (prod_rec x r) @ (iter r)
  in
  iter l
;;

let decision p =

  (* essai d'une paire x,y de clauses : *)

  let rec essai ((x,y),l) =
    let lr = resolutions x y in
    if succes lr then
      true
    else
      exists (fun r -> derive (r::l)) lr
```



```
  (* derivation a partir d'une liste de clause : *)

  and derive = function
      []  -> false
    | [c] -> false
    | l   -> let combi = paires l in exists essai combi

  in

  let negp = Neg p in
  let fc   = fnc negp in
  derive fc
;;

(* Complexite :
 *
 * Soit n clauses de taille <= t.
 * Alors C(n) = n(n-1)/2 x t x C(n-1), d'ou
 *
 * C(n) = (n!)^2 x t^n / (2^n x n) ~ A x (B x n)^2n (Stirling)
 *
 * Comparaison avec les tables de verite :
 *
 * Le nombre de prop. atomiques p de P se situe entre log2(n) et n
 * et la methode des tables de verites est alors en 4^p.
 * Elle est donc plus efficace, mais la methode de resolution
 * peut-etre utilisee pour la logique de premier ordre et meme au dela
 * (semi-decidabilite seulement !).
 *)

(* Club Ecossais : *)

let E = Atome "E"  (* est ecossais *)
and C = Atome "C"  (* porte des chaussettes rouges *)
and K = Atome "K"  (* porte un kilt *)
and M = Atome "M"  (* est marie *)
and D = Atome "D"  (* sort le dimanche *) ;;

let r1 = Imp (Neg E, C)
and r2 = Ou (K, Neg C)
and r3 = Imp (M, Neg D)
and r4 = Et (Imp (D, E) , Imp (E,D))
and r5 = Imp (K, Et (E, M))
and r6 = Imp (E, K) ;;

let pas_de_membre =
  Imp (r1 , Imp (r2, Imp(r3 ,Imp (r4, Imp(r5, Imp (r6, Faux)))))) ;;

(* #decision pas_de_membre;;
 * - : bool = true                                               *)
```