```
type proposition = Const of bool
                         | Var of char
                         | Neg of proposition
                         | Et of proposition * proposition
                         | Ou of proposition * proposition
                         | Imp of proposition * proposition ;;

type ifexpression = Cst of bool
                          | Vr of char
                          | If of ifexpression * ifexpression * ifexpression
;;

type assignation == (char * bool) list ;;

type resultat = Tautologie | Refutation of assignation ;;
```

(* question 1 *)

```
let rec prop_to_if = function
  | Const c -> Cst c
  | Var s -> Vr s
  | Neg p -> If (prop_to_if p, Cst false, Cst true)
  | Et (p, q) -> If (prop_to_if p, prop_to_if q, Cst false)
  | Ou (p, q) -> If (prop_to_if p, Cst true, prop_to_if q)
  | Imp (p, q) -> If (prop_to_if p, prop_to_if q, Cst true) ;;

let p1 = Imp (Et (Var `a`, Var `b`), Var `a`)
and p2 = Imp (Var `a`, Et (Var `a`, Var `b`))
and p3 = Imp (Imp (Var `a`, Imp (Var `a`, Var `b`)), Var `b`) ;;

prop_to_if p1 ;;
(* if (if a then b else Faux) then a else Vrai *)
prop_to_if p2 ;;
(* if a then (if a then b else Faux) else Vrai *)
prop_to_if p3 ;;
(* if (if a then (if a then b else Vrai) else Vrai) then b else Vrai *)
```

(* question 2 *)

```
let rec est_normale = function
  | Cst _ -> true
  | Vr _ -> true
  | If (Vr _, p, q) -> est_normale p && est_normale q
  | _ -> false ;;
```

(* question 3 *)

```
let rec normalise = function
  | Cst b -> Cst b
  | Vr s -> Vr s
  | If (Cst true, p, q) -> normalise p
  | If (Cst false, p, q) -> normalise q
  | If (Vr s, p, q) -> If (Vr s, normalise p, normalise q)
  | If (If (m, p, q), r, s) -> normalise (If (m, normalise (If (p, r, s)),
normalise (If (q, r, s)))) ;;

normalise (prop_to_if p1) ;;
(* if a then (if b then a else Vrai) else Vrai *)
```

```
normalise (prop_to_if p2) ;;
(* if a then (if a then b else Faux) then Vrai *)
normalise (prop_to_if p3) ;;
(* if a then (if a then (if b then b else Vrai) else b) else b *)
```

(* question 4 *)

```
let rec decision_partielle alpha = function
  | Cst true -> Tautologie
  | Cst false -> Refutation alpha
  | Vr s -> (try let b = assoc s alpha in if b then Tautologie else
Refutation alpha
                with Not_found -> Refutation ((s, false)::alpha))
  | If (Vr s, p, q) -> (try let b = assoc s alpha in
                                    if b then decision_partielle alpha p
else decision_partielle alpha q
                                  with Not_found -> let alphat = (s,
true)::alpha and alphaf = (s, false)::alpha in
                                                    let d1 =
decision_partielle alphat p
                                                    and d2 =
decision_partielle alphaf q
                                                    in match (d1,
d2) with
                                                       |
Tautologie, Tautologie -> Tautologie
                                                       | Refutation
beta, Tautologie -> Refutation beta
                                                       |
Tautologie, Refutation beta -> Refutation beta
                                                       | Refutation
beta1, Refutation beta2 -> Refutation beta1 )
  | _ -> failwith "decision_partielle" ;;

let decision p = decision_partielle [] (normalise (prop_to_if p)) ;;

decision p1 ;;
(* resultat = Tautologie *)
decision p2 ;;
(* resultat = Refutation [`b`, false; `a`, true] *)
decision p3 ;;
(* resultat = Refutation [`b`, false; `a`, false] *)
```