
Formules logiques et BDD

Projet IAP 1

Groupe 3.1
Mme DUBOIS

7 novembre 2006

Ronan LE TOULLEC

Table des matières

1	Introduction	4
2	Arbres de décision	4
2.1	Problème	4
2.2	Résolution	4
2.2.1	Idée générale	4
2.2.2	Définition des types	5
2.2.3	La fonction principale	5
2.2.4	Extraire la liste des variables	5
2.2.5	Evaluer une formule	7
3	<i>Binary Decision Diagram</i>	7
3.1	Problème : Construction	7
3.2	Résolution : Construction	8
3.2.1	Idée générale	8
3.2.2	Définition des types	8
3.2.3	La fonction principale	8
3.2.4	Placer les feuilles qui seront dans le <i>BDD</i>	8
3.2.5	Construire le <i>BDD</i>	9
3.2.6	Rechercher les adresses	11
3.3	Problème : Simplification	11
3.4	Résolution : Simplification	12
3.4.1	Idée générale	12
3.4.2	La fonction principale	12
3.4.3	Simplifier partiellement le <i>BDD</i>	12
3.5	Deux fonctions supplémentaires	13
3.5.1	<code>bdd_of_dec</code>	13
3.5.2	<code>bddSimple_of_formule</code>	13
4	Affichage graphique	14
4.1	Problème	14
4.2	Résolution : <i>Binary Decision Diagrams</i>	14
4.2.1	Idée générale	14
4.2.2	La fonction	14
4.3	Résolution : Arbres de décision	15
4.3.1	Idée générale	15
4.3.2	Définition des types	15
4.3.3	La fonction principale	15
4.3.4	Numérotter un arbre de décision	16
5	Etudier les formules	16
5.1	Problème : Tautologie	16
5.2	Résolution : Tautologie	16
5.2.1	Idée générale	16
5.2.2	La fonction	17
5.3	Problème : Équivalence	17
5.4	Résolution : Équivalence	17
5.4.1	Idée générale	17
5.4.2	La fonction principale	17
5.4.3	Construire deux arbres de décision avec la même liste de variables	18
5.4.4	Modifier les adresses	19
5.5	Problème : Satisfiabilité	19
5.6	Résolution : Satisfiabilité	20
5.6.1	Idée générale	20
5.6.2	La fonction principale	20

5.6.3	Renvoyer toutes les assignations rendant vraie la formule	20
6	Conclusion	21
A	Les tests¹	22
A.1	Arbres de décision	22
A.1.1	isole_var	22
A.1.2	appartient	22
A.1.3	elimine_doublons	22
A.1.4	variables	22
A.1.5	valeur_logique	23
A.1.6	eval	23
A.1.7	dec_of_formule	23
A.2	BDD	24
A.2.1	adresse	24
A.2.2	construitBDD	24
A.2.3	appartientArbre	26
A.2.4	depart	26
A.2.5	bdd_of_formule	26
A.2.6	simplifie	27
A.2.7	bddSimplePartiel	27
A.2.8	bddSimple_of_bdd	27
A.2.9	bdd_of_dec	28
A.2.10	bddSimple_of_formule	28
A.3	Affichages graphiques	29
A.3.1	dotBDD	29
A.3.2	nbelements	29
A.3.3	numerote	29
A.3.4	dotDEC	30
A.4	Etude des formules	30
A.4.1	tautologie	30
A.4.2	doubleDEC	31
A.4.3	remplace	31
A.4.4	formules_equivalentes	32
A.4.5	assignations_vraies	32
A.4.6	assignation_vraie	32
B	Le code	33

¹ Afin de ne pas surcharger le corps du rapport, les tests n'apparaîtront que dans cette annexe.

1 Introduction

On cherche à représenter de manière simple des propositions logiques, afin de les étudier : une formule est-elle une tautologie ? Est-elle satisfiable ? Deux formules, sont-elles équivalentes ? Grâce à une représentation par des arbres, on pourra efficacement déterminer ces caractéristiques. Les formules seront exprimées à l'aide de cinq opérateurs, un opérateur unaire :

- le **Non** (\neg);

et quatre opérateurs binaires :

- le **Et** (\vee);

- le **Ou** (\wedge);

- le **Implique** (\Rightarrow);

- et le **Est_équivalent** (\Leftrightarrow);

ainsi que par l'utilisation de *variables*, dont le nom sera libre, et dont la valeur logique sera soit **true**, soit **false**.

Afin d'étudier les formules, celles-ci seront représentées par des *arbres de décision*. C'est-à-dire des arbres dont pour un noeud donné, l'*étiquette* est une variable de la formule, le *fil gauche* est l'arbre de décision obtenu en remplaçant cette variable par **true** et le *fil droit* l'arbre de décision obtenu en remplaçant la variable par **false**. Une fois que toutes les variables ont été traitées, viennent les *feuilles* qui valent **true** si la proposition est Vraie, et **false** dans le cas contraire.

Ensuite, ces arbres de décision seront simplifiés en *Binary Decision Diagrams*, c'est à dire en leur supprimant tous les sous-arbres identiques, et en créant plusieurs redirections vers les mêmes nouveaux noeuds. Un noeud pourra ainsi avoir plusieurs prédécesseurs.

Dans un premier temps, nous construirons les arbres de décision. Puis, dans une deuxième partie, nous les transformerons en des *Binary Decision Diagrams*. Enfin, nous travaillerons sur ces représentations, d'abord en les représentant graphiquement, puis en déterminant si les formules initiales étaient des tautologies, satisfiables, ou bien si deux formules étaient équivalentes.

2 Arbres de décision

2.1 Problème

À partir d'une formule, on cherche à construire l'arbre de décision correspondant.

```
(* Interface : dec_of_formule
   Type : tformule -> arbreDec
   Argument : formule
   Precondition : aucune
   Postcondition : renvoie l'arbre de decision correspondant a la formule donnee
*)
```

2.2 Résolution

2.2.1 Idée générale

Petit à petit, on va chercher à construire des environnements associant les variables à leur valeur booléenne. Ainsi, on va commencer par extraire les noms des variables de la formule. Ensuite, en parcourant cette liste, il suffira de créer des noeuds contenant :

- pour étiquette le nom de la variable;

- pour fil gauche le sous-arbre créé en remplaçant la variable par **true**;

- et pour sous-arbre droit le sous-arbre créé en remplaçant la variable par **false**.

Enfin, quand toutes les variables auront été traitées, on reverra une feuille contenant l'évaluation de la formule logique dans son environnement.

2.2.2 Définition des types

D'abord, définissons le type des formules :

```
type tformule =
  | Valeur of bool
  | Var of string
  | Non of tformule
  | Et of tformule * tformule
  | Ou of tformule * tformule
  | Implique of tformule * tformule
  | Est_equivalent of tformule * tformule
;;
```

Définissons ensuite le type des arbres de décision :

```
type arbreDec =
  | FeuilleDec of bool
  | Racine of string * arbreDec * arbreDec
;;
```

Ainsi, un `arbreDec` est composé soit d'une `FeuilleDec` contenant un booléen, soit d'une `Racine` contenant une étiquette, un sous-arbre gauche, et un sous-arbre droit.

Les environnements seront des listes de couple : `string * bool list`.

Les noms des variables seront stockées dans une liste : `string list`.

2.2.3 La fonction principale

```
let dec_of_formule formule =
  let rec aux liste de variable, environnement =
    match liste de variable with
    | [] -> FeuilleDec (evaluation de la formule dans l'environnement)
    | t::q ->Racine(le nom de la variable = t,
                    on reapplique <<aux>>
                    sur le reste de la liste de variables
                    en ajoutant (t,true) a l'environnement,
                    on reapplique <<aux>>
                    sur le reste de la liste de variables
                    en ajoutant (t,false) a l'environnement,
                    )
  in
  aux (liste des variables de la formule) (environnement initial = vide)
;;
```

On constate qu'il nous faut une fonction permettant de récupérer le nom des variables de la formule, et une fonction nous permettant d'évaluer une formule grâce à un environnement donné. Soient `variables` et `eval` ces fonctions.

Il vient donc :

```
let dec_of_formule formule =
  let rec aux var_liste environnement =
    match var_liste with
    | [] -> FeuilleDec (eval formule environnement)
    | t::q -> Racine (t,
                      aux q ((t,true)::environnement),
                      aux q ((t,false)::environnement)
                    )
  in
  aux (variables formule) []
;;
```

2.2.4 Extraire la liste des variables

```

(* Interface variables
Type : tformule -> string list
Argument : formule
Precondition : aucune
Postcondition : renvoie la liste des variable contenues dans la formule,
                sans doublons.
                si la formule ne contient pas de variables, renvoie la liste vide
*)

```

```

let variables formule = elimine_doublons (isole_var formule) ;;

```

La fonction `elimine_doublons` supprimera les doublons, et la fonction `isole_var` extraira la liste des variables d'une formule, en autorisant les doublons. Ce découpage en deux fonctions se justifie par le besoin de réutiliser la fonction `elimine_doublons` ultérieurement.

```

(* Interface elimine_doublons
Type : 'a list -> 'a list
Argument : liste
Precondition : aucune
Postcondition : supprime les doublons d'une liste
*)

```

```

let rec elimine_doublons l =
  match l with
  | [] -> []
  | t::q -> if appartient t q then ( elimine_doublons q )
            else t::( elimine_doublons q )
;;

```

La fonction `elimine_doublons` fait référence à la fonction `appartient`, voici son interface et son code :

```

(* Interface appartient
Type : 'a -> 'a list -> bool
Argument : element, liste
Precondition : aucune
Postcondition : renvoie true si l'element est dans la liste, false sinon.
*)

```

```

let rec appartient e l =
  match l with
  | [] -> false
  | t::q -> ( t = e ) || ( appartient e q )
;;

```

Dans la suite, cette fonction `appartient` sera réutilisée. Enfin, voici la fonction `isole_var`

```

(* Interface isole_var
Type : tformule -> string list
Argument : formule
Precondition : aucune
Postcondition : renvoie la liste des variable contenue dans la formule,
                avec des doublons possibles
                si la formule ne contient pas de variables, renvoie la liste vide
*)

```

```

let rec isole_var formule =
  match formule with
  | Valeur _ -> []
  | Var x -> [x]
  | Non f -> isole_var f
  | Et (f1,f2)
  | Ou (f1,f2)

```

```

    | Implique (f1,f2)
    | Est_equivalent (f1,f2) -> ( isole_var f1 ) @ ( isole_var f2 )
;;

```

2.2.5 Evaluer une formule

La fonction `eval` est utile pour évaluer la valeur logique d'une formule, dans un environnement donné.

```

(* Interface eval
  Type : tformule -> (string * bool) list -> bool
  Argument : formule, environnement
  Precondition : l'environnement doit englober toutes les variables de la formule
  Postcondition : renvoie la valeur de verite de la formule en donnant
                  aux variables la valeur qui leur correspond dans l'environnement
*)

```

```

let rec eval formule env =
  match formule with
  | Valeur x -> x
  | Var p -> valeur_logique p env
  | Non a -> not (eval a env)
  | Et (a,b) -> (eval a env) && (eval b env)
  | Ou (a,b) -> (eval a env) || (eval b env)
  | Implique (a,b) -> (not (eval a env)) || (eval b env)
  | Est_equivalent (a,b) -> ((eval a env) && (eval b env))
                          || ((not (eval a env)) && (not (eval b env)))
;;

```

cette est celle qui donne leur sens aux connecteurs. elle utilise la fonction `valeur_logique` :

```

(* Interface valeur_logique
  Type : string -> string * bool list -> bool
  Argument : nom de variable, environnement
  Precondition : la variable doit etre dans l'environnement
  Postcondition : renvoie la valeur de verite de la variable dans l'environnement
  Raises : failwith "valeur_logique : la variable n'est pas dans l'environnement !"
*)

```

```

let rec valeur_logique p environnement=
  match environnement with
  | [] -> failwith "valeur_logique : la variable n'est pas dans l'environnement !"
  | (t,b)::q -> if t = p then b
                 else valeur_logique p q
;;

```

3 *Binary Decision Diagram*

3.1 Problème : Construction

Dans un arbre de décision, on peut retrouver plusieurs fois les mêmes branches. Aussi, quand deux nœuds ont des sous-arbres fils identiques, on souhaiterait qu'ils se dirigent réellement vers le même successeur, en partageant ce dernier.

```

(* Interface bdd_of_formule
  Type : tformule -> bddNoeud list
  Argument : formule
  Precondition : aucune
  Postcondition : renvoie le BDD correspondant a la formule
*)

```

3.2 Résolution : Construction

3.2.1 Idée générale

À chaque étape de construction il s'agit de vérifier dans le graphe déjà construit si le nœud que l'on désire ajouter est déjà présent ou non. S'il est présent, on le réutilise en récupérant son numéro, sinon on l'ajoute à la liste. Pour ce, on va commencer par convertir la formule en arbre de décision, puis, on va construire (récursivement) un *Binary Decision Diagram* et une liste d'arbres de décision **en même temps**, qui représenteront tous deux la liste des sous-arbres déjà ajoutés dans le *Binary Decision Diagram*. Pour que chaque sous-arbre soit correctement identifié, on se basera sur la longueur du *BDD* déjà construit.

Le principe consiste à ne jamais dissocier le *Binary Decision Diagram* déjà construit, et la liste d'arbres de décision qui lui correspond, car c'est justement grâce à cette liste que l'on va déterminer si un sous-arbre est déjà présent dans le *BDD* ou pas.

3.2.2 Définition des types

Le type d'un nœud d'un *Binary Decision Diagram* sera le suivant :

```
type bddNoeud =
  | FeuilleBdd of int * bool
  | NoeudAutre of int * string * int * int
;;
```

et les *Binary Decision Diagrams* seront donc des `bddNoeud * list`. Un *BDD* sera donc une liste de `bddNoeud` comportant soit une `FeuilleBdd` contenant une adresse et une valeur booléenne, soit un `NoeudAutre` contenant une adresse, une étiquette, une adresse de successeur gauche, et une adresse de successeur droit.

3.2.3 La fonction principale

La fonction `bdd_of_formule` doit renvoyer un *Binary Decision Diagram*. Or, d'après ce que l'on a déjà dit, on va construire, en même temps, le *Binary Decision Diagram* et la liste d'arbres de décision. Comme seul le *Binary Decision Diagram* nous intéresse, on ne renverra qu'une seule composante d'un programme qui créera les deux en même temps. De plus, ce programme devra avoir pour argument un arbre de décision, et non une formule, on emploiera donc la fonction `dec_of_formule`. Enfin, afin d'avoir les feuilles à la fin de la `bddNoeud * list`, on commencera à construire les arbres en préinsérant les feuilles qui existeront dans le graphe. Ainsi, les adresses 1 et 2 seront réservées respectivement à `true` et `false`. Il vient donc :

```
let bdd_of_formule formule =
  let arbreDEC = dec_of_formule formule
  in
    let departBDD, departDEC = depart arbreDEC
    in
      fst (construitBDD arbreDEC departBDD departDEC)
;;
```

avec `construitBDD` et `depart`, les fonctions définies ci-dessous.

3.2.4 Placer les feuilles qui seront dans le *BDD*

Pour des raisons pratiques, on va toujours placer les `feuilleBdds` à la fin des listes de `bddNoeuds`. On va donc regarder dans l'arbre de décision quelles sont les feuilles qui seront utiles, et lancer le programme de construction sur des liste contenant déjà ces feuilles.

```
(* Interface depart
  Type : arbreDec -> bddNoeud list * arbreDec list
  Argument : arbre de decision
  Precondition : aucune
  Postcondition : renvoie le couple compose des bddNoeuds utilises,
                  puis de la liste des arbreDec utilises.
*)
```

```

let depart dec =
  if appartientArbre (FeuilleDec (true)) dec
  then if appartientArbre (FeuilleDec (false)) dec
    then [FeuilleBdd (2, false); FeuilleBdd (1, true)],
         [FeuilleDec (false); FeuilleDec (true)]
    else [FeuilleBdd (1, true)], [FeuilleDec (true)]
  else if appartientArbre (FeuilleDec (false)) dec
    then [FeuilleBdd (2, false)], [FeuilleDec (false)]
    else [], [] (* pour exhaustivite! *)
;;

```

On a besoin d'une fonction qui nous dira si une feuille est dans un arbre ou pas. On crée donc la fonction `appartientArbre` :

```

(* Interface appartientArbre
   Type : arbreDec -> arbreDec -> bool
   Argument : arbre de decision (element), arbre de decision (environnement)
   Precondition : on recherche un element qui est une feuille
   Postcondition : teste si l'element est dans l'environnement
*)

```

```

let rec appartientArbre element arbre =
  match arbre with
  | Racine (_, arbreG, arbreD) -> (appartientArbre element arbreG)
                                || (appartientArbre element arbreD)
  | _ -> element = arbre
;;

```

3.2.5 Construire le BDD

```

(* Interface construitBDD
   Type : arbreDec -> bddNoeud list -> arbreDec list
                                               -> bddNoeud list * arbreDec list
   Argument : arbre qu'on veut transformer,
             bdd deja cree,
             liste des sous-arbres deja ajoutees au bdd,
             compteur qui servira a adresser les bddNoeuds
   Precondition : a chaque element dans la liste du bdd,
                 correspond son equivalent dans la liste des sous-arbres.
                 Les feuilles doivent etre deja inserees
                 dans le BDD et la liste des sous-arbres.
   Postcondition : renvoie le bdd correspondant a l'arbre a construire,
                 et la liste des sous-arbres.
                 N'insere pas les feuilles.
*)

```

Expliquons cette fonction pas à pas. D'abord, repérons les arguments :

- `arbreDec` représente l'arbre de décision à transformer ;
- `bdd` est le *Binary Decision Diagram* que l'on construit peu à peu, ce que l'on renverra ;
- `donnees` représente la liste d'arbres de décision que l'on construit en même temps que le *Binary Decision Diagram*. Il s'appelle `donnees` car, il «contient les données» du *BDD*.

Pour donner une adresse aux BDD noeuds, il suffira de compter les éléments de la liste des noeuds.

«Vu de loin», l'algorithme ressemble à cela :

```

let rec construitBDD arbreDec bdd donnees =
  match arbreDec with
  | FeuilleDec -> on ne fait rien, car les feuilles sont deja presentes
                 dans les donnees, par preconstruction
                 on renvoie donc le bdd et les donnees
  | Racine (nom, sous-arbre gauche, sous-arbre droit) ->

```

```

on regarde si l'arbreDec est dans les donnees.
si oui, on ne fait rien, on renvoie le bdd et et les donnees
sinon on regarde si les sous-arbres gauches et/ou droits sont
dans les donnees.
s'il n'y sont pas, on les insere, puis on renvoie le NoeudAutre
ayant pour fils les adresses des sous-arbres gauches et droits,
et l'arbreDec ayant pour fils les sous-arbres gauches et droits.
;;

```

Rapprochons-nous :

```

let rec construitBDD arbreDec bdd donnees =
match arbreDec with
| FeuilleDec _ -> on renvoie de bdd et les donnees
| Racine (nom,sous-arbre G,sous-arbre D) ->
on regarde si l'arbreDec est dans les donnees.
si oui, on ne fait rien, on renvoie le bdd et et les donnees
sinon si les sous-arbres G et D sont dans les donnees
alors on ajoute (NoeudAutre(adresse,nom,adresse du sous-arbre G,
adresse du sous-arbre D)) au bdd,
et Racine(nom,sous-arbre G,sous-arbre D) aux donnees
sinon si le sous-arbre G est dans les donnees mais pas le D
alors on calcule
(bddD,donneesD) = construitBDD du sous-arbreD bdd donnees
et on ajoute (NoeudAutre(adresse,nom,adresse du sous-arbre G,
adresse du sous-arbre D) au bddD
et Racine(nom,sous-arbre G,sous-arbre D)aux donneesD
sinon si le sous-arbre D est dans les donnees mais pas le G
alors on calcule
(bddG,donneesG) = construitBDD du sous-arbreG bdd donnees
et on ajoute (NoeudAutre(adresse,nom,adresse du sous-arbre G,
adresse du sous-arbre D) au bddG
et Racine(nom,sous-arbre G,sous-arbre D)aux donneesG
sinon si aucun des sous-arbres n'est dans les donnees
alors on calcule
(bddG,donneesG) = construitBDD du sous-arbreG bdd donnees
et (bddD,donneesD) = construitBDD du sous-arbreD
sur les donnees qu'on vient d'obtenir: bddG et donneesG
et on ajoute (NoeudAutre(adresse,nom,adresse du sous-arbre G,
adresse du sous-arbre D) au bddD
et Racine(nom,sous-arbre G,sous-arbre D)aux donneesD
;;

```

On voit clairement apparaître le besoin d'avoir une fonction *adresse* qui renverrait l'adresse d'un *NoeudBdd*, et qui prendrait donc en argument un arbre à rechercher, un *Binary Decision Diagram*, et une liste d'arbres de décision.

Enfin, il vient :

```

let rec construitBDD arbreDec bdd donnees=
match arbreDec with
| FeuilleDec _ -> (bdd,donnees)
| Racine (nom,arbreG,arbreD) ->
if appartient arbreDec donnees
then (bdd,donnees)
else if appartient arbreG donnees
then if appartient arbreD donnees
then (NoeudAutre(1+(List.length bdd),nom,adresse arbreG bdd donnees,
adresse arbreD bdd donnees)::bdd,
Racine(nom,arbreG,arbreD)::donnees)
else let (bddD,donneesD) = construitBDD arbreD bdd donnees in
(NoeudAutre(1+(List.length bdd),nom,adresse arbreG bdd donnees,
adresse arbreD bdd donneesD)::bddD,
Racine(nom,arbreG,arbreD)::donneesD)

```

```

else if appartient arbreD donnees
then let (bddG,donneesG) = construitBDD arbreG bdd donnees in
      (NoeudAutre(1+(List.length bddG),nom,adresse arbreG bddG donneesG,
                adresse arbreD bdd donnees)::bddG,
      Racine(nom,arbreG,arbreD)::donneesG)
else let (bddG,donneesG) = construitBDD arbreG bdd donnees in
      let (bddD,donneesD) = construitBDD arbreD bddG donneesG in
          (NoeudAutre(1+(List.length bddD),nom,adresse arbreG bddD donneesD,
                    adresse arbreD bddD donneesD)::bddD,
          Racine(nom,arbreG,arbreD)::donneesD)
;;

```

Cette fonction réutilise la fonction `appartient` décrite page 6.

3.2.6 Rechercher les adresses

```

(* Interface adresse
Type: arbreDec -> bddNoeud list -> arbreDec list -> int
Argument : arbre recherche,
           bdd (qui contient l'adresse),
           liste d'arbres de decision
Precondition : a chaque element dans la liste du bdd,
               correspond son equivalent dans la liste des sous-arbres
               l'ordre des elements doit etre le meme dans les 2 listes,
               l'arbreDec recherche doit etre dans la liste d'arbres de decision
Postcondition : renvoie l'adresse de l'arbre recherche
Raises : failwith "adresse: l'arbre n'est pas dans les donnees".
         failwith "adresse: le bdd et le dec ne coincident pas".
*)

let rec adresse arbre bdd donnees =
  match bdd,donnees with
  | [] , [] -> failwith "adresse: l'arbre n'est pas dans les donnees"
  | (FeuilleBdd (numero,p))::bddReste , (FeuilleDec q)::donneesReste ->
      if FeuilleDec p = arbre then numero
      else adresse arbre bddReste donneesReste
  | (NoeudAutre (numero,nom,g,d))::bddReste , (Racine(q,g2,d2))::donneesReste ->
      if (Racine (nom,g2,d2)) = arbre then numero
      else adresse arbre bddReste donneesReste
  | _ , _ -> failwith "adresse: le bdd et le dec ne coincident pas"
;;

```

La précondition voulant que le BDD et la liste d'arbres de décision soient toujours équivalents est assurée par construction. Quand on agit sur une liste (parcours, ajout d'un élément...) alors on le fait toujours sur les deux listes en même temps.

3.3 Problème : Simplification

Afin d'améliorer encore cette structure de données, on peut supprimer les nœuds dont deux les successeurs sont identiques. On obtient ainsi des *BDD simplifiés*. Il s'agit donc de supprimer des nœuds.

```

(* Interface bddSimple_of_bdd
Type : bddNoeud list -> bddNoeud list
Argument : BDD
Precondition : aucune
Postcondition : renvoie le BDD simplifie correspondant au BDD
*)

```

3.4 Résolution : Simplification

3.4.1 Idée générale

On va parcourir le *BDD*, pour trouver un nœud dont les deux successeurs ont la même adresse. Si on en trouve un, alors on le simplifiera. On réappliquera cette fonction jusqu'à ce qu'il n'y ait plus de nœuds à simplifier.

Pour simplifier un nœud, on se munira d'un *BDD*, de l'adresse qu'il faut remplacer, et de la nouvelle adresse. En parcourant le *BDD* on changera toutes les occurrences de l'ancienne adresse. Enfin, on pensera à supprimer le *NoeudAutre* ayant pour adresse, l'adresse à changer.

3.4.2 La fonction principale

Nous «fixerons» la fonction en usant de la même technique que la fonctionnelle *fixe*, mais nous appliquerons directement son principe dans la fonction principale. Cette fonction sera appelée *bddSimple_of_bdd* et la fonction partielle *bddSimplePartiel*. On aura donc :

```
let rec bddSimple_of_bdd bdd =
  let bdd2 = (bddSimplePartiel bdd bdd) in
    if bdd2 = bdd then bdd else bddSimple_of_bdd bdd2
;;
```

3.4.3 Simplifier partiellement le *BDD*

```
(* Interface bddSimplePartiel
  Type : bddNoeud list -> bddNoeud list -> bddNoeud list
  Argument : BDD, BDD bis
  Precondition : BDD = BDD bis
  Postcondition : renvoie le BDD en le simplifiant partiellement
                  ne touche pas aux feuilles
*)
```

Cette fonction admet deux fois le même argument car tandis que l'on en parcourra un, l'autre sera celui sur lequel on appliquera effectivement les modifications.

```
let rec bddSimplePartiel bdd bdd2 =
  match bdd with
  | NoeudAutre (adresse,nom,adresseG,adresseD)::q ->
    if adresseG = adresseD
    then simplifie bdd2 adresse adresseG
    else bddSimplePartiel q bdd2
  | _ -> bdd2
    (* Si feuille ou [], alors on a fini, on renvoie bdd2 *)
;;
```

Enfin, il vient la fonction *simplifie* qui va simplifier proprement dit le *BDD*.

```
(* Interface simplifie
  Type : bddNoeud list -> int -> int -> bddNoeud list
  Argument : BDD, valeur a changer, direction
  Precondition : aucune
  Postcondition : renvoie le BDD en modifiant les valeurs a changer
                  par la direction
                  ne touche pas aux feuilles
*)
```

Voici comment fonctionne de cette fonction :

```
let rec simplifie bdd2 achanger direction =
  match bdd2 with
  | NoeudAutre (adresse,nom,g,d)::q ->
    si l'adresse du NoeudAutre est achanger,
    alors on supprime le NoeudAutre, et on traite le reste q de la liste.
```

```

        sinon si g et/ou d est achanger, on le/les remplace par la direction,
        puis on traite q.
        si ni g et ni d ne sont achanger, alors on les laisse et on traite q.
    | _ -> quand on a une feuille, on a fini, on les renvoie.
;;

```

L'algorithme est donc :

```

let rec simplifie bdd2 achanger direction =
  match bdd2 with
  | NoeudAutre (adresse,nom,g,d)::q ->
    if adresse = achanger then simplifie q achanger direction else
    if g = achanger
    then if d = achanger
    then simplifie q achanger direction
    else NoeudAutre (adresse,nom,direction,d)
    ::(simplifie q achanger direction)
    else if d = achanger
    then NoeudAutre (adresse,nom,g,direction)
    ::(simplifie q achanger direction)
    else NoeudAutre (adresse,nom,g,d)
    ::(simplifie q achanger direction)
  | _ -> bdd2
;;

```

3.5 Deux fonctions supplémentaires

3.5.1 bdd_of_dec

Munissons-nous dès à présent d'une fonction qui nous permettra d'afficher un *BDD simplifié* à partir d'un arbre de décision. Il suffit en fait de modifier la fonction `bdd_of_formule` en `bdd_of_dec` :

```

(* Interface bdd_of_dec
  Type : arbreDec -> bddNoeud list
  Argument : arbre de decision
  Precondition : aucune
  Postcondition : renvoie le BDD simplifie correspondant a l'arbre de decision
*)

let bdd_of_dec arbreDEC =
  let departBDD,departDEC = depart arbreDEC
  in
  bddSimple_of_bdd (fst (construitBDD arbreDEC departBDD departDEC))
;;

```

3.5.2 bddSimple_of_formule

On va définir une fonction qui nous permettra de convertir directement une formule en *BDD simplifié*.

```

(* Interface bddSimple_of_formule
  Type : tformule -> bddNoeud list
  Argument : formule
  Precondition : aucune
  Postcondition : renvoie le BDD simplifie correspondant a la formule
*)

let bddSimple_of_formule formule =
  bddSimple_of_bdd (bdd_of_formule formule)
;;

```

4 Affichage graphique

4.1 Problème

On cherche maintenant à afficher graphiquement des *Binary Decision Diagrams* et des arbres de décision. Pour ce, on utilise un programme externe DOT qui permet d'afficher des graphes à partir de fichiers dot. Il faut programmer des fonctions qui produiront des fichiers dot à partir d'arbres de décision, afin qu'en lançant le programme, nous obtenions graphiquement les graphes souhaités. Nous devons donc produire les deux fonctions suivantes :

```
(* Interface dotBDD
   Type : arbreDec -> unit
   Argument : Arbre de decision
   Precondition : aucune
   Postcondition : renvoie un fichier graphique dot representant
                   le BDD qui correspond a l'arbre de decision
*)
et
(* Interface dotDEC
   Type : arbreDec -> unit
   Argument : Arbre de decision
   Precondition : aucune
   Postcondition : renvoie un fichier graphique dot representant
                   l'arbre de decision
*)
```

Enfin, on souhaite que tous les arcs indiquants les successeurs gauches soient en pointillés de couleur rouge, les successeurs droits en traits pleins noirs, et que les feuilles soient en caractères gras.

4.2 Résolution : *Binary Decision Diagrams*

4.2.1 Idée générale

Pour produire un fichier dot, il faut ouvrir un fichier en écriture sous OCaml, puis écrire dedans dans un langage qui sera compréhensible par le programme DOT. Le nom du fichier de sortie sera «dotBDDfichier.dot». Une fois que l'on aura écrit dans le fichier, il faudra le fermer. La syntaxe est la suivante :

```
fichier = open_out "dotBDDfichier.dot";
  on ecrit dans le fichier tout ce qu'on veut
close_out fichier;;
```

Le contenu du fichier doit commencer par «digraph G {\n», le \n sert à passer à la ligne. Il doit se terminer par «}». On va construire récursivement le corps du fichier. D'abord, chaque nœud doit avoir un numéro distinct. On utilisera les adresses. Ensuite, ce numéro doit apparaître sur trois lignes. Dans la première, on trouvera l'étiquette du nœud, dans la deuxième, l'adresse du successeur gauche, et dans la troisième, celle du droit. Afin de respecter les styles des feuilles et des arcs, derrière le numéro du successeur gauche, on ajoutera «[color=red,style=dashed] ;\n», et derrière les feuilles «[style = bold,label= la valeur logique de la feuille] ;\n».

4.2.2 La fonction

```
let dotBDD arbre =
  let fichier = open_out "dotBDDfichier.dot"
  in
  let bdd = bdd_of_dec arbre
  in
  let rec enForme bdd =
    match bdd with
    | [] -> ""
```

```

| FeuilleBdd (numero,p)::q ->
  ( (string_of_int numero) ^ " [style = bold, label=\\""
  ^ (if p then "V" else "F") ^ "\";\n" ) ^ enForme q
| NoeudAutre (numero,nom,g,d)::q ->
  (string_of_int numero) ^ " [ label=\\""^nom(*^" : "
  ^ (string_of_int numero)* ^ "\";\n"
  ^ (string_of_int numero) ^ " -> " ^ (string_of_int g)
  ^ " [color=red,style=dashed];\n"
  ^ (string_of_int numero) ^ " -> " ^ (string_of_int d) ^ ";\n"
  ^ enForme q
in
  output_string fichier ("digraph G {\n" ^ (enForme bdd) ^ "}");
  close_out fichier
;;

```

4.3 Résolution : Arbres de décision

4.3.1 Idée générale

Le principe est exactement le même que précédemment. Les seules différences sont que l'on ne parcourra non plus des listes, mais des arbres, et que les nœuds ne sont plus numérotés. Il faudra donc les numéroté par ailleurs.

4.3.2 Définition des types

On va définir le type `arbreDecNum`. les `arbreDecNums` seront des `arbreDecs` auxquels on aura rajouter des numéros uniques aux noeuds et aux feuilles. On obtient donc :

```

type arbreDecNum =
| FeuilleDecNum of int*bool
| RacineNum of int*string*arbreDecNum*arbreDecNum
;;

```

4.3.3 La fonction principale

La fonction principale suivra exactement le même schémas que la fonction précédente, mais filtrera des arbres, et non des listes.

```

let dotDEC arbredec =
  let fichier = open_out "dotDECfichier.dot"
  in
    let rec enForme arbredecnum =
      match arbredecnum with
      | FeuilleDecNum (numero,p) -> (string_of_int numero)
        ^ " [style = bold, label=\\""
        ^ (if p then "V" else "F") ^ "\";\n"
      | RacineNum (numero,nom,FeuilleDecNum (numG,p),FeuilleDecNum (numD,q)) ->
        (string_of_int numero)^" [ label=\\""^nom^"\";\n"
        ^ (string_of_int numero)^" -> "^(string_of_int numG)
        ^" [color=red,style=dashed];\n"
        ^ (string_of_int numero)^" -> "^(string_of_int numD)^";\n"
        ^ enForme (FeuilleDecNum (numG,p))
        ^ enForme (FeuilleDecNum (numD,q))
      | RacineNum (numero,nom,RacineNum (numG,g,gg,gd),RacineNum (numD,d,dg,dd)) ->
        (string_of_int numero)^" [ label=\\""^nom^"\";\n"
        ^ (string_of_int numero)^" -> "^(string_of_int numG)
        ^" [color=red,style=dashed];\n"
        ^ (string_of_int numero)^" -> "^(string_of_int numD)^";\n"
        ^ enForme (RacineNum (numG,g,gg,gd))
        ^ enForme (RacineNum (numD,d,dg,dd))
      | _ -> failwith "DotDec : Pour exhaustivite"
    in

```

```

        output_string fichier ("digraph G {" ^ (enForme (numerote arbredec)) ^ "}");
        close_out fichier
;;

```

On utilise la fonction `numerote` définie ci-après, pour numéroter les arbres.

4.3.4 Numéroter un arbre de décision

```

(* Interface numerote
   Type : arbreDec -> arbreDecNum
   Argument : arbre de decision
   Precondition : aucune
   Postcondition : attribue une adresse distincte a chaque racine ou feuille
                   de l'arbre de decision
*)

let numerote arbredec =
  let rec aux arbre compteur =
    match arbre with
    | FeuilleDec p -> FeuilleDecNum (compteur,p)
    | Racine (p,g,d) -> RacineNum (compteur,p,aux g (compteur+1),
                                   aux d (compteur+1+(nbelements g)))
  in
    aux arbredec 1
;;

```

Pour utiliser cette fonction, on a besoin de connaître le nombre d'éléments dans un arbre, et on doit donc créer la fonction `nbelements` qui va compter les éléments d'un arbre. La voici :

```

(* Interface nbelements
   Type : arbreDec -> int
   Argument : arbre de decision
   Precondition : aucune
   Postcondition : renvoie le nombre de Racines et FeuilleDecs de l'arbre de decision
*)

let rec nbelements a =
  match a with
  | FeuilleDec _ -> 1
  | Racine (_,g,d) -> 1 + (nbelements g) + (nbelements d)
;;

```

5 Etudier les formules

5.1 Problème : Tautologie

On dispose d'une formule logique, et on souhaite savoir si cette formule est une tautologie. Dans ce cas, on renverra `true`, dans le cas contraire, on renverra `false`.

```

(* Interface tautologie
   Type : tformule -> bool
   Argument : formule
   Precondition : aucune
   Postcondition : teste si la formule est une tautologie
*)

```

5.2 Résolution : Tautologie

5.2.1 Idée générale

Si une formule est une tautologie, alors son arbre de décision ne comporte que des feuilles `true`. Ainsi, son *Binary Decision Diagram* ne possède pas de feuille `false`. Les successeurs des racines

les plus basses sont donc tous la `feuilleBdd(1,true)`. Une fois simplifié, il ne reste plus que cette feuille. Ainsi, quand une formule est une tautologie, son *Binary Decision Diagram* se résume à une feuille `true`.

5.2.2 La fonction

La fonction est donc simple à construire :

```
let tautologie form =
  bddSimple_of_formule form = [FeuilleBdd(1,true)]
;;
```

5.3 Problème : Équivalence

On se donne deux formules, et on veut savoir si elles sont équivalente ou non, c'est-à-dire si elles ont le même tableau de vérité ou non.

```
(* Interface formules_equivalentes
   Type : tformule -> tformule -> bool
   Argument : formule1, formule2
   Precondition : aucune
   Postcondition : teste si les formules sont equivalentes
*)
```

5.4 Résolution : Équivalence

5.4.1 Idée générale

On admettra que deux formules sont équivalentes si et seulement si les deux *BDDs simplifiés* construits avec le même ordre pour les variables sont identiques, au numérotage des noeuds près. Il est donc nécessaire que les deux *BDDs simplifiés* aient le même nombre d'éléments.

Une fois que l'on aura construit les deux *BDDs simplifiés* à partir du même ordre de variables, on va parcourir les deux *BDDs* en même temps, tout en se munissant d'un double du deuxième *BDD* qui servira de référence et dans lequel on changera les adresses. Pour avoir les mêmes adresses dans les deux *BDDs*, on va donc parcourir les deux en même temps, changer les adresses du premier, regarder ce qu'il faut changer dans le deuxième, et effectuer les modifications dans le double du deuxième. À la fin, on comparera le premier et le double du deuxième.

5.4.2 La fonction principale

```
let formules_equivalentes form1 form2 =
  let rec aux bdd1 bdd2 bdd2fixe securite=
    match bdd1,bdd2 with
    | [],[] -> cas de base, on renvoie bdd1 et bdd2fixe
    | NoeudAutre (adresse1,nom1,adresseG1,adresseD1)::q1,
      NoeudAutre (adresse2,nom2,adresseG2,adresseD2)::q2 ->
      On ajoute aux adresses du premier noeud la securite,
      pour avoir de nouvelles adresses libres.
      Ensuite, remplace les adresses du deuxieme noeud:
      On remplace chaque occurrence de l'adresse2 dans le bdd2fixe,
      par (adresse1+securite), et simplement dans ce noeud,
      adresseG2 par adresseG1+securite et
      adresseD2 par adresseD1+securite.
      Cependant, si les noms des noeuds sont differents, alors les
      arbres sont differents, et on arrete tout,
      on renvoie les deux arbres,
    | _ -> Dans tous les autres cas, on ne fait rien.
      Si on n'a pas deux feuilleBdds, alors tant pis,
      les arbres seront differents,
```

```

        sinon, on n'a pas besoin de modifier les feuilles
        pour savoir s'il sont identiques.
in
    On va construire les deux arbres de decision avec la meme liste de variables.
    Puis, on va en deduire les BDD correspondants.
    On va definir la securite egale a la longueur de bdd2.
    Et enfin, on comparera les deux BDDs obtenus par la
    fonctions <<aux>> appliquee a tous ces elements.
;;

    On voit qu'il faudra une fonction remplace qui fait exactement la même chose que la fonction
    simplifie, mais sans supprimer d'éléments. Enfin, on aboutit à cette solution :

let formules_equivalentes form1 form2 =
  let rec aux bdd1 bdd2 bdd2fixe securite=
    match bdd1,bdd2 with
    | NoeudAutre (adresse1,nom1,adresseG1,adresseD1)::q1,
      NoeudAutre (adresse2,nom2,adresseG2,adresseD2)::q2 ->
      if nom1=nom2 then
        let nouveauBDD1,nouveauBDD2fixe =
          aux q1 q2 (remplace bdd2fixe (adresse2) (adresse1+securite)
            (adresseG1+securite) (adresseD1+securite))
            securite
        in
          (NoeudAutre(adresse1+securite,nom1,ardesseG1+securite,
            adresseD1+securite)::nouveauBDD1,
            nouveauBDD2fixe)
        else (bdd1,bdd2fixe)
    | _ -> (bdd1,bdd2fixe)
  in
  let arbredec1,arbredec2 = doubleDEC form1 form2
  in
  let bdd1,bdd2 = (bdd_of_dec arbredec1,bdd_of_dec arbredec2)
  in
  let securite = List.length bdd2
  in
  let (nouveauBDD1,nouveauBDD2) = aux (bdd1) (bdd2) (bdd2) (securite)
  in nouveauBDD1 = nouveauBDD2
;;

```

On va donc créer deux nouvelles fonctions : remplace et doubleDEC.

5.4.3 Construire deux arbres de décision avec la même liste de variables

```

(* Interface doubleDEC
Type : tformule -> tformule -> arbreDec * arbreDec
Argument : formule1, formule2
Precondition : aucune
Postcondition : renvoie les arbres de decision correspondant
                aux formules donnees,
                en utilisant la meme liste de variables
*)

```

C'est une condition nécessaire pour déterminer l'équivalence entre les deux formules. En fait, cette fonction est la même que `dec_of_formule`, sauf que l'on va construire deux arbres et non plus un seul. Il faudra cependant veiller à concatener les deux listes de variables, avant d'éliminer les doublons, avec la fonction `elimine_doublons` définie page 6, pour ne pas tomber dans l'exception `failwith "valeur_logique : la variable n'est pas dans l'environnement!"` dans le cas où des variables seraient dans la deuxième formule, et pas dans la première.

```

let doubleDEC form1 form2 =
  let rec aux form var_liste environnement =
    match var_liste with
    | [] -> FeuilleDec (eval form environnement)

```

```

    | t::q -> Racine (t,
                      aux form q ((t,true)::environnement),
                      aux form q ((t,false)::environnement)
                    )
  in
    let liste_variables = elimine_doublons ((variables form1)@(variables form2))
    in
      ( aux form1 (liste_variables) [] , aux form2 (liste_variables) [] )
;;

```

5.4.4 Modifier les adresses

Maintenant, on veut parcourir le deuxième *BDD* pour changer toutes les adresses, afin d'uniformiser la numérotation avec celle du premier *BDD*. Le principe de cette fonction sera exactement le même que celui de la fonction `simplifie`, sauf qu'elle ne supprimera pas d'éléments, et qu'elle remplacera en plus les adresses des successeurs gauches et droits par celles du premier *BDD*, après avoir ajouté la sécurité, valeur égale au nombre d'éléments du deuxième *BDD*, afin de ne pas réattribuer des numéros déjà existants.

```

(* Interface remplace
   Type : bddNoeud list -> int -> int -> int -> int -> bddNoeud list
   Argument : BDD dont on ne doit pas modifier la taille
               valeur de l'adresse a changer
               valeur par laquelle on doit remplacer l'adresse
               nouvelle adresse du successeur gauche
               nouvelle adresse du successeur droit
   Precondition : aucune
   Postcondition : renvoie le BDD en modifiant toutes les occurrences
                   de l'adresse a changer par la nouvelle adresse.
                   Et dans le noeud dont on a change l'adresse,
                   on remplace les successeurs gauches et droits
                   par les nouvelles adresses des successeurs.
                   ne modifie pas les feuilles.
*)

let rec remplace bdd2fixe achanger direction nouveauG nouveauD =
  match bdd2fixe with
  | NoeudAutre (adresse,nom,g,d)::q ->
    if adresse = achanger
    then (NoeudAutre (direction,nom,nouveauG,nouveauD))
         ::remplace q achanger direction nouveauG nouveauD
    else
    if g = achanger
    then if d = achanger
         then remplace q achanger direction nouveauG nouveauD
         else NoeudAutre (adresse,nom,direction,d)
              ::(remplace q achanger direction nouveauG nouveauD)
    else if d = achanger
         then NoeudAutre (adresse,nom,g,direction)
              ::(remplace q achanger direction nouveauG nouveauD)
         else NoeudAutre (adresse,nom,g,d)
              ::(remplace q achanger direction nouveauG nouveauD)
  | _ -> bdd2fixe
;;

```

5.5 Problème : Satisfiabilité

Maintenant, on désire renvoyer une assignation rendant vraie une formule, si elle est satisfiable. Sinon, on échouera.

```

(* Interface assignation_vraie

```

```

Type : tformule -> (string * bool) list
Argument : formule
Precondition : la formule doit etre satisfiable
Postcondition : renvoie une assignation assignation rendant la formule vraie
Raises : failwith "assignation_vraie : la formule n'est pas satisfiable"
*)

```

5.6 Résolution : Satisfiabilité

5.6.1 Idée générale

Pour résoudre ce problème, on va utiliser les arbres de décision. En parcourant l'arbre, on saura la formule est satisfiable, puis il suffira de remonter cet arbre afin de connaître toutes les assignations rendant vraie la formule.

Une assignation correspondra en fait à un **environnement**, c'est-à-dire une liste de couples `string*bool`, qui associera à chaque variable sa valeur booléenne.

5.6.2 La fonction principale

Ainsi, on utilisera une autre fonction, nommée `assignations_vraies`, pour renvoyer une liste contenant toutes les assignations qui rendent vraie la formule. Si cette liste est vide, alors c'est que la formule n'est pas satisfiable. On échouera.

```

let assignation_vraie formule =
  match (assignations_vraies formule) with
  | [] -> failwith "assignation_vraie : la formule n'est pas satisfiable"
  | t::q -> t
;;

```

5.6.3 Renvoyer toutes les assignations rendant vraie la formule

Maintenant, on va rechercher ces assignations.

```

(* Interface assignations_vraies
Type : tformule -> (string * bool) list list
Argument : formule
Precondition : aucune
Postcondition : renvoie les assignations assignation rendant la formule vraie,
ou la liste vide s'il n'en existe pas
*)

```

On va réutiliser le principe de la fonction `dec_of_formule`, à savoir créer un environnement récursivement pour chaque feuille. Ensuite, il suffira d'évaluer la formule dans l'environnement créé, et si cette évaluation est `true`, alors on renverra l'environnement. On obtiendra ainsi une liste des assignations rendant vraie la formule.

```

let assignations_vraies formule =
  let rec aux var_liste environnement =
    match var_liste with
    | [] -> if (eval formule environnement) then [environnement] else []
    | t::q -> (aux q ((t,true)::environnement))@(aux q ((t,false)::environnement))
  in
  aux (variables formule) []
;;

```

6 Conclusion

En conclusion, nous avons d'abord défini les arbres de décision et les *Binary Decision Diagrams*, puis créé une fonction permettant de les construire à partir de formules booléennes. Ensuite, nous les avons représentées graphiquement avant de chercher à savoir si ces formules étaient satisfiables, des tautologies, ou encore si deux formules étaient équivalentes.

Cependant, on peut remarquer que certaines fonctions pourraient être optimisées, ou implémentées de manières totalement différentes. Ainsi, quand on considère que deux fonctions sont équivalentes, on compare ici leur *Binary Decision Diagrams*. Mais on pourrait aussi tester si le `est_Equivalent` de ces deux fonctions est une tautologie, grâce à la fonction créée juste avant.

On voit donc que le choix de la méthode d'implantation dépend beaucoup des contraintes imposées, de la complexité du programme, et... des préférences du programmeur.

Enfin, il est à souligner que si cette représentation est satisfaisante pour l'esprit, elle reste peu pratique. De manière générale, les connecteurs sont utilisés de manière infixée. Or, partout dans ce projet, ils sont utilisés de manière préfixée, ce qui est moins naturel, et ne facilite pas la lecture.

De même, tester des fonctions s'avère délicat quand les arguments sont trop longs, car cela prend une place énorme, et qu'il faut créer des arbres parfois très complexes. C'est pourquoi, dans ce rapport, les tests ont tous été regroupés dans une annexe spécialement dédiée à cela, et que les arbres y sont aussi courts que possible.

Pour terminer, ajoutons enfin que ce projet aura été pour moi l'occasion de découvrir `LATEX`, pendant la phase de rédaction du rapport.

A Les tests

Par souci de lisibilité, les tests ne seront visibles que dans cette annexe. Pour chaque fonction, nous allons donner les tests effectués, et le résultat de ces tests.

A.1 Arbres de décision

A.1.1 isole_var

On essaie une formule avec des doublons.

```
# isole_var (Est_equivalent(Et(Non(Var "p"),Var "q"),Implique(Var "r",Var "p")));;
- : string list = ["p"; "q"; "r"; "p"]
```

On essaie une formule sans variables.

```
# isole_var (Est_equivalent(Non(Valeur true),Ou(Valeur false,Valeur true)));;
- : string list = []
```

A.1.2 appartient

On essaie un cas où l'élément appartient à la liste.

```
# appartient 1 [1;5;4;1;6;1;5;4;8;9;3;2;5;7];;
- : bool = true
```

On essaie un cas où l'élément n'appartient pas à la liste.

```
# appartient 2 [];;
- : bool = false
```

On essaie sur une liste d'un autre type.

```
# appartient 10.4 [1.73;2.25;5.97;4.5;9.6;8.12;6.84;7.04;3.0];;
- : bool = false
```

A.1.3 elimine_doublons

On essaie sur une liste avec des doublons.

```
# elimine_doublons [1;5;4;1;6;1;5;4;8;9;3;2;5;7];;
- : int list = [6; 1; 4; 8; 9; 3; 2; 5; 7]
```

On essaie sur la liste vide.

```
# elimine_doublons [];;
- : 'a list = []
```

On essaie sur une liste d'un autre type.

```
# elimine_doublons [1.;2.;5.;4.;9.;8.;6.;7.;3.];;
- : float list = [1.; 2.; 5.; 4.; 9.; 8.; 6.; 7.; 3.]
```

A.1.4 variables

On essaie sur une formule où des variables apparaissent plusieurs fois.

```
# variables (Est_equivalent(Et(Non(Var "p"),Var "q"),Implique(Var "q",Var "p")));;
- : string list = ["q"; "p"]
```

On essaie sur une formule sans variable.

```
# variables (Est_equivalent(Et(Non(Valeur true),Valeur false),
                             Implique(Valeur false,Valeur true))));;
- : string list = []
```

On essaie sur une formule sans doublons.

```
# variables (Est_equivalent(Implique(Non(Var "p"),Valeur true),Et(Var "r",Var "s")));;
- : string list = ["p"; "r"; "s"]
```

A.1.5 valeur_logique

On essaie un cas où la variable recherchée vaut false.

```
# valeur_logique "p" [("q", true); ("p", false); ("r", true)];;
- : bool = false
```

On essaie un cas où la variable recherchée vaut true.

```
# valeur_logique "r" [("q", true); ("p", false); ("r", true)];;
- : bool = true
```

On essaie un cas où la variable recherchée n'est pas dans l'environnement (précondition non respectée).

```
# valeur_logique "s" [("q", true); ("p", false); ("r", true)];;
Exception: Failure "valeur_logique : la variable n'est pas dans l'environnement !".
```

A.1.6 eval

On essaie un cas où l'évaluation doit être true.

```
# eval (Et(Or(Valeur false, Var "p"), Implique(Var "q", Var "r")))
      [("q", true); ("p", true); ("r", true)];;
- : bool = true
```

On essaie un cas où l'évaluation doit être false.

```
# eval (Est_equivalent(Et(Non(Var "p"), Var "q"), Implique(Var "q", Var "p")))
      [("q", true); ("p", false); ("r", true)];;
- : bool = false
```

On essaie un cas où il y a des variables inutiles dans l'environnement.

```
# eval (Est_equivalent(Et(Non(Valeur true), Valeur false), Valeur true))
      [("q", true); ("p", false); ("r", true)];;
- : bool = false
```

On essaie quand une variable n'est pas présente dans l'environnement (précondition non respectée).

```
# eval (Est_equivalent(Et(Non(Var "p"), Var "q"), Implique(Var "q", Var "p")))
      [("s", true); ("p", false); ("r", true)];;
Exception: Failure "valeur_logique : la variable n'est pas dans l'environnement !".
```

A.1.7 dec_of_formule

On essaie sur la formule de l'énoncé.

```
# dec_of_formule (
  Et(Est_equivalent(Var "P1", Var "Q1"), Est_equivalent(Var "P2", Var "Q2")));;
- : arbreDec =
  Racine ("P1",
    Racine ("Q1",
      Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
        Racine ("Q2", FeuilleDec false, FeuilleDec true)),
      Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
        Racine ("Q2", FeuilleDec false, FeuilleDec false))),
    Racine ("Q1",
      Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
        Racine ("Q2", FeuilleDec false, FeuilleDec false)),
      Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
        Racine ("Q2", FeuilleDec false, FeuilleDec true))))
```

On essaie sur une autre formule.

```
# dec_of_formule (Ou(Et(Var "p",Non(Var "q")),Var "r"));;
- : arbreDec =
  Racine ("p",
    Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec false),
      Racine ("r", FeuilleDec true, FeuilleDec true)),
    Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec false),
      Racine ("r", FeuilleDec true, FeuilleDec false)))
```

On essaie sur une formule sans variable.

```
# dec_of_formule (Est_equivalent(Et(Non(Valeur true),Valeur false),Valeur true));;
- : arbreDec = FeuilleDec false
```

A.2 BDD

A.2.1 adresse

On essaie en recherchant une Racine.

```
# adresse (Racine("p",FeuilleDec false,FeuilleDec true))
  [NoeudAutre (3, "p", 2, 1); FeuilleBdd (2, false); FeuilleBdd (1, true)]
  [(Racine("p",FeuilleDec false,FeuilleDec true));
   FeuilleDec false;
   FeuilleDec true];;
- : int = 3
```

On essaie en recherchant une FeuilleDec.

```
# adresse (FeuilleDec false)
  [NoeudAutre (3, "p", 2, 1); FeuilleBdd (2, false); FeuilleBdd (1, true)]
  [(Racine("p",FeuilleDec false,FeuilleDec true));
   FeuilleDec false;
   FeuilleDec true];;
- : int = 2
```

On essaie avec un *BDD* plus long que la listes des arbres de décision.

```
# adresse (Racine("p",FeuilleDec true,FeuilleDec true))
  [NoeudAutre (3, "p", 2, 1); FeuilleBdd (2, false); FeuilleBdd (1, true)]
  [FeuilleDec true];;
Exception: Failure "adresse: le bdd et le dec ne coincident pas".
```

On essaie en recherchant une Racine qui n'est pas dans les données.

```
# adresse (Racine("p",FeuilleDec true,FeuilleDec true))
  [NoeudAutre (3, "p", 2, 1); FeuilleBdd (2, false); FeuilleBdd (1, true)]
  [(Racine("p",FeuilleDec false,FeuilleDec true));
   FeuilleDec false;
   FeuilleDec true];;
Exception: Failure "adresse: l'arbre n'est pas dans les donnees".
```

A.2.2 construitBDD

On essaie sur l'arbre de décision de la formule de l'énoncé.

```
# construitBDD
(Racine ("P1",
  Racine ("Q1",
    Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
      Racine ("Q2", FeuilleDec false, FeuilleDec true)),
    Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
      Racine ("Q2", FeuilleDec false, FeuilleDec false))),
  Racine ("Q1",
    Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
```

```

        Racine ("Q2", FeuilleDec false, FeuilleDec false)),
        Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
        Racine ("Q2", FeuilleDec false, FeuilleDec true))))
[FeuilleBdd (2, false); FeuilleBdd (1, true)]
[FeuilleDec false; FeuilleDec true];

- : bddNoeud list * arbreDec list =
([NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 7, 5);
NoeudAutre (8, "Q1", 5, 7); NoeudAutre (7, "P2", 6, 6);
NoeudAutre (6, "Q2", 2, 2); NoeudAutre (5, "P2", 3, 4);
NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
FeuilleBdd (2, false); FeuilleBdd (1, true)],
[Racine ("P1",
Racine ("Q1",
Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec true)),
Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec false))),
Racine ("Q1",
Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec false)),
Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec true))));
Racine ("Q1",
Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec false)),
Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec true))));
Racine ("Q1",
Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec true)),
Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec false)));
Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec false));
Racine ("Q2", FeuilleDec false, FeuilleDec false);
Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
Racine ("Q2", FeuilleDec false, FeuilleDec true));
Racine ("Q2", FeuilleDec false, FeuilleDec true);
Racine ("Q2", FeuilleDec true, FeuilleDec false); FeuilleDec false;
FeuilleDec true])

```

On essaie sur un arbre ne contenant que des FeuilleDecs true

```

# construitBDD
(Racine ("p",
Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
Racine ("r", FeuilleDec true, FeuilleDec true)),
Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
Racine ("r", FeuilleDec true, FeuilleDec true))))
[FeuilleBdd (1, true)]
[FeuilleDec true];

```

```

- : bddNoeud list * arbreDec list =
([NoeudAutre (4, "p", 3, 3); NoeudAutre (3, "q", 2, 2);
NoeudAutre (2, "r", 1, 1); FeuilleBdd (1, true)],
[Racine ("p",
Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
Racine ("r", FeuilleDec true, FeuilleDec true)),
Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
Racine ("r", FeuilleDec true, FeuilleDec true))));
Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),

```

```

    Racine ("r", FeuilleDec true, FeuilleDec true));
Racine ("r", FeuilleDec true, FeuilleDec true); FeuilleDec true]]

```

La fonction ne sait pas si la condition n'est pas respectée. Mais elle pourra renvoyer des résultats faux dans ce cas.

A.2.3 appartientArbre

On teste le cas où on recherche une feuille qui n'est pas dans l'arbre.

```

# appartientArbre (FeuilleDec false) (Racine ("p",
    Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
        Racine ("r", FeuilleDec true, FeuilleDec true)),
    Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
        Racine ("r", FeuilleDec true, FeuilleDec true))));
- : bool = false

```

On teste le cas où la feuille est dans l'arbre.

```

# appartientArbre (FeuilleDec false)
(Racine ("P1",
    Racine ("Q1",
        Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
            Racine ("Q2", FeuilleDec false, FeuilleDec true)),
        Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
            Racine ("Q2", FeuilleDec false, FeuilleDec false))),
    Racine ("Q1",
        Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
            Racine ("Q2", FeuilleDec false, FeuilleDec false)),
        Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
            Racine ("Q2", FeuilleDec false, FeuilleDec true)))););
- : bool = true

```

A.2.4 depart

On teste le cas où on ne renvoie qu'une feuille.

```

# depart (Racine ("p",
    Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
        Racine ("r", FeuilleDec true, FeuilleDec true)),
    Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
        Racine ("r", FeuilleDec true, FeuilleDec true)))););
- : bddNoeud list * arbreDec list =
([FeuilleBdd (1, true)], [FeuilleDec true])

```

On teste le cas standard.

```

# depart (Racine ("p",
    Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
        Racine ("r", FeuilleDec true, FeuilleDec false)),
    Racine ("q", Racine ("r", FeuilleDec false, FeuilleDec true),
        Racine ("r", FeuilleDec true, FeuilleDec true)))););
- : bddNoeud list * arbreDec list =
([FeuilleBdd (2, false); FeuilleBdd (1, true)],
 [FeuilleDec false; FeuilleDec true])

```

A.2.5 bdd_of_formule

On essaie sur la formule de l'énoncé.

```
# bdd_of_formule (
  Et(Est_equivalent(Var "P1",Var "Q1"),Est_equivalent(Var "P2",Var "Q2")));;

- : bddNoeud list =
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 7, 5);
 NoeudAutre (8, "Q1", 5, 7); NoeudAutre (7, "P2", 6, 6);
 NoeudAutre (6, "Q2", 2, 2); NoeudAutre (5, "P2", 3, 4);
 NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
 FeuilleBdd (2, false); FeuilleBdd (1, true)]

  On teste avec une tautologie.

# bdd_of_formule (Ou(Var "p",Non(Var "p")));;
- : bddNoeud list = [NoeudAutre (2, "p", 1, 1); FeuilleBdd (1, true)]
```

A.2.6 simplifie

On essaie le cas utile qui servira : en supprimant un nœud qui a deux successeurs identiques. On prend le cas de l'énoncé.

```
# simplifie
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 7, 5);
 NoeudAutre (8, "Q1", 5, 7); NoeudAutre (7, "P2", 6, 6);
 NoeudAutre (6, "Q2", 2, 2); NoeudAutre (5, "P2", 3, 4);
 NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
 FeuilleBdd (2, false); FeuilleBdd (1, true)]
7 6;;
- : bddNoeud list =
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 6, 5);
 NoeudAutre (8, "Q1", 5, 6); NoeudAutre (6, "Q2", 2, 2);
 NoeudAutre (5, "P2", 3, 4); NoeudAutre (4, "Q2", 2, 1);
 NoeudAutre (3, "Q2", 1, 2); FeuilleBdd (2, false); FeuilleBdd (1, true)]
```

A.2.7 bddSimplePartiel

On teste le cas de l'énoncé.

```
# bddSimplePartiel
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 7, 5);
 NoeudAutre (8, "Q1", 5, 7); NoeudAutre (7, "P2", 6, 6);
 NoeudAutre (6, "Q2", 2, 2); NoeudAutre (5, "P2", 3, 4);
 NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
 FeuilleBdd (2, false); FeuilleBdd (1, true)]
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 7, 5);
 NoeudAutre (8, "Q1", 5, 7); NoeudAutre (7, "P2", 6, 6);
 NoeudAutre (6, "Q2", 2, 2); NoeudAutre (5, "P2", 3, 4);
 NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
 FeuilleBdd (2, false); FeuilleBdd (1, true)];;
- : bddNoeud list =
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 6, 5);
 NoeudAutre (8, "Q1", 5, 6); NoeudAutre (6, "Q2", 2, 2);
 NoeudAutre (5, "P2", 3, 4); NoeudAutre (4, "Q2", 2, 1);
 NoeudAutre (3, "Q2", 1, 2); FeuilleBdd (2, false); FeuilleBdd (1, true)]
```

A.2.8 bddSimple_of_bdd

On essaie avec l'exemple de l'énoncé.

```
# bddSimple_of_bdd
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 7, 5);
 NoeudAutre (8, "Q1", 5, 7); NoeudAutre (7, "P2", 6, 6);
```

```

    NoeudAutre (6, "Q2", 2, 2); NoeudAutre (5, "P2", 3, 4);
    NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
    FeuilleBdd (2, false); FeuilleBdd (1, true)];;
- : bddNoeud list =
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 2, 5);
 NoeudAutre (8, "Q1", 5, 2); NoeudAutre (5, "P2", 3, 4);
 NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
 FeuilleBdd (2, false); FeuilleBdd (1, true)]

```

On essaie avec le *BDD* d'une tautologie.

```

# bddSimple_of_bdd [NoeudAutre (2, "p", 1, 1); FeuilleBdd (1, true)];;
- : bddNoeud list = [FeuilleBdd (1, true)]

```

A.2.9 bdd_of_dec

On essaie avec l'exemple de l'énoncé.

```

# bdd_of_dec
(Racine ("P1",
  Racine ("Q1",
    Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
      Racine ("Q2", FeuilleDec false, FeuilleDec true)),
    Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
      Racine ("Q2", FeuilleDec false, FeuilleDec false))),
  Racine ("Q1",
    Racine ("P2", Racine ("Q2", FeuilleDec false, FeuilleDec false),
      Racine ("Q2", FeuilleDec false, FeuilleDec false)),
    Racine ("P2", Racine ("Q2", FeuilleDec true, FeuilleDec false),
      Racine ("Q2", FeuilleDec false, FeuilleDec true)))));;
- : bddNoeud list =
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 2, 5);
 NoeudAutre (8, "Q1", 5, 2); NoeudAutre (5, "P2", 3, 4);
 NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
 FeuilleBdd (2, false); FeuilleBdd (1, true)]

```

On teste sur l'arbre de décision d'une tautologie.

```

# bdd_of_dec (Racine ("p",
  Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
    Racine ("r", FeuilleDec true, FeuilleDec true)),
  Racine ("q", Racine ("r", FeuilleDec true, FeuilleDec true),
    Racine ("r", FeuilleDec true, FeuilleDec true))));;
- : bddNoeud list = [FeuilleBdd (1, true)]

```

A.2.10 bddSimple_of_formule

On essaie avec l'exemple de l'énoncé.

```

# bddSimple_of_formule (
  Et(Est_equivalent(Var "P1",Var "Q1"),Est_equivalent(Var "P2",Var "Q2")));;
- : bddNoeud list =
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 2, 5);
 NoeudAutre (8, "Q1", 5, 2); NoeudAutre (5, "P2", 3, 4);
 NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
 FeuilleBdd (2, false); FeuilleBdd (1, true)]

```

On teste sur une tautologie.

```

# bddSimple_of_formule (Ou(Var "p",Non(Var "p")));;
- : bddNoeud list = [FeuilleBdd (1, true)]

```

A.3 Affichages graphiques

A.3.1 dotBDD

On essaie de créer l'arbre de décision donnée dans l'énoncé.

```
# dotBDD (  
  dec_of_formule (  
    Et(Est_equivalent(Var "P1",Var "Q1"),Est_equivalent(Var "P2",Var "Q2"))));  
- : unit = ()
```

On trouve le graphe 1 :

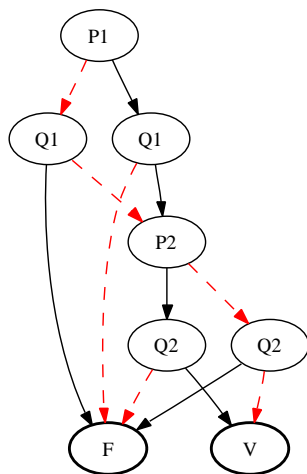


FIG. 1 – BDD de l'énoncé

On teste sur une tautologie.

```
# dotBDD (dec_of_formule (Ou(Var "p",Non(Var "p"))));  
- : unit = ()
```

On trouve le graphe 2 :

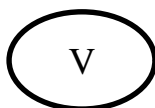


FIG. 2 – BDD d'une tautologie

A.3.2 nbelements

On essaie sur l'exemple de l'énoncé.

```
# nbelements (dec_of_formule (  
  Et(Est_equivalent(Var "P1",Var "Q1"),Est_equivalent(Var "P2",Var "Q2"))));  
- : int = 31
```

On compte le nombre d'éléments d'une feuille.

```
# nbelements (dec_of_formule (Valeur false));  
- : int = 1
```

A.3.3 numerote

On numérote les éléments de l'arbre de décision de l'énoncé.

```
# numerote (dec_of_formule (
  Et(Est_equivalent(Var "P1",Var "Q1"),Est_equivalent(Var "P2",Var "Q2"))));
- : arbreDecNum =
RacineNum (1, "P1",
  RacineNum (2, "Q1",
    RacineNum (3, "P2",
      RacineNum (4, "Q2", FeuilleDecNum (5, true), FeuilleDecNum (6, false)),
      RacineNum (7, "Q2", FeuilleDecNum (8, false), FeuilleDecNum (9, true))),
    RacineNum (10, "P2",
      RacineNum (11, "Q2", FeuilleDecNum (12, false), FeuilleDecNum (13, false)),
      RacineNum (14, "Q2", FeuilleDecNum (15, false), FeuilleDecNum (16, false))),
  RacineNum (17, "Q1",
    RacineNum (18, "P2",
      RacineNum (19, "Q2", FeuilleDecNum (20, false), FeuilleDecNum (21, false)),
      RacineNum (22, "Q2", FeuilleDecNum (23, false), FeuilleDecNum (24, false))),
    RacineNum (25, "P2",
      RacineNum (26, "Q2", FeuilleDecNum (27, true), FeuilleDecNum (28, false)),
      RacineNum (29, "Q2", FeuilleDecNum (30, false), FeuilleDecNum (31, true))))))
```

On numérote une feuille.

```
# numerote (dec_of_formule (Valeur false));
- : arbreDecNum = FeuilleDecNum (1, false)
```

A.3.4 dotDEC

On essaie avec l'arbre de décision de l'énoncé.

```
# dotDEC (dec_of_formule (
  Et(Est_equivalent(Var "P1",Var "Q1"),Est_equivalent(Var "P2",Var "Q2"))));
- : unit = ()
```

On trouve le graphe 3 :

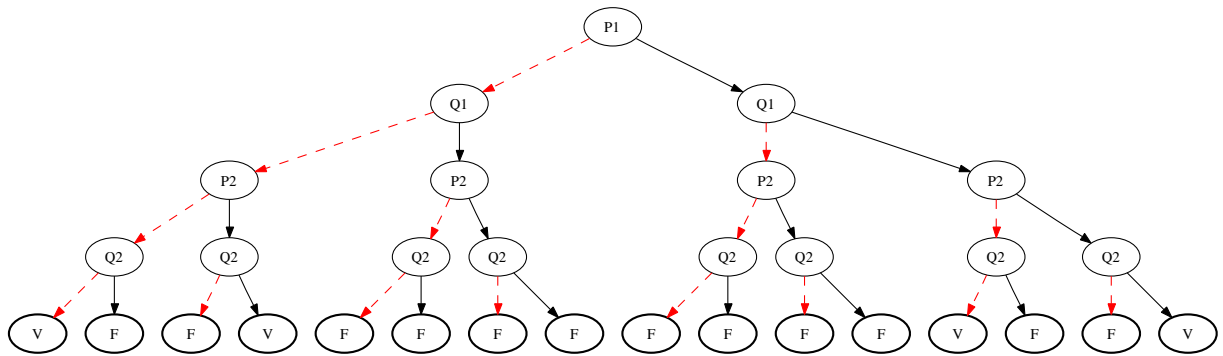


FIG. 3 – Arbre de décision de l'énoncé

On essaie avec un autre arbre.

```
# dotDEC (dec_of_formule (Ou(Var "p",Non(Var "p"))));
- : unit = ()
```

On trouve le graphe 4 page suivante :

A.4 Etude des formules

A.4.1 tautologie

On teste un cas où la formule n'est pas une tautologie.

```
# tautologie (Est_equivalent(Et(Non(Var "p"),Var "q"),Implique(Var "r",Var "p")));
- : bool = false
```

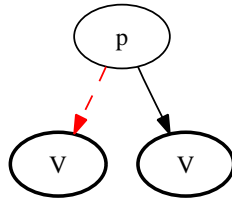


FIG. 4 – Arbre de décision d’une tautologie

On teste un cas où la formule est une tautologie.

```
# tautologie (Ou(Var "p",Non(Var "p")));;
- : bool = true
```

A.4.2 doubleDEC

On teste un cas général.

```
# doubleDEC (Et(Ou(Var "P1",Var "Q1"),Var "P2"))
              (Ou(Et(Var "P1",Var "P2"),Et(Var "Q1",Var "P2"))));;
```

```
- : arbreDec * arbreDec =
(Racine ("r",
  Racine ("q", Racine ("p", FeuilleDec true, FeuilleDec true),
    Racine ("p", FeuilleDec true, FeuilleDec true)),
  Racine ("q", Racine ("p", FeuilleDec true, FeuilleDec true),
    Racine ("p", FeuilleDec false, FeuilleDec false))),
Racine ("r",
  Racine ("q", Racine ("p", FeuilleDec true, FeuilleDec true),
    Racine ("p", FeuilleDec false, FeuilleDec true)),
  Racine ("q", Racine ("p", FeuilleDec true, FeuilleDec true),
    Racine ("p", FeuilleDec false, FeuilleDec true))))
```

On teste un cas où les variables sont différentes entre les deux formules.

```
# doubleDEC (Implique(Non(Var "r"),Var "q"))(Ou(Var "q",Non(Var "p")));;

- : arbreDec * arbreDec =
(Racine ("P1",
  Racine ("Q1", Racine ("P2", FeuilleDec true, FeuilleDec false),
    Racine ("P2", FeuilleDec true, FeuilleDec false)),
  Racine ("Q1", Racine ("P2", FeuilleDec true, FeuilleDec false),
    Racine ("P2", FeuilleDec false, FeuilleDec false))),
Racine ("P1",
  Racine ("Q1", Racine ("P2", FeuilleDec true, FeuilleDec false),
    Racine ("P2", FeuilleDec true, FeuilleDec false)),
  Racine ("Q1", Racine ("P2", FeuilleDec true, FeuilleDec false),
    Racine ("P2", FeuilleDec false, FeuilleDec false))))
doubleDEC (Ou(Non(Valeur true),Valeur true))(Et(Valeur true,Non(Valeur false)));;
```

On teste un cas sans variables.

```
# doubleDEC (Ou(Non(Valeur true),Valeur true))(Et(Valeur true,Non(Valeur false)));;

- : arbreDec * arbreDec = (FeuilleDec true, FeuilleDec true)
```

A.4.3 remplace

On teste un cas général.

```

# remplace
  [NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 2, 5);
   NoeudAutre (8, "Q1", 5, 2); NoeudAutre (5, "P2", 3, 4);
   NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
   FeuilleBdd (2, false); FeuilleBdd (1, true)]
  5 16 14 17 ;;
- : bddNoeud list =
[NoeudAutre (10, "P1", 8, 9); NoeudAutre (9, "Q1", 2, 16);
 NoeudAutre (8, "Q1", 16, 2); NoeudAutre (16, "P2", 14, 17);
 NoeudAutre (4, "Q2", 2, 1); NoeudAutre (3, "Q2", 1, 2);
 FeuilleBdd (2, false); FeuilleBdd (1, true)]

```

A.4.4 formules_équivalentes

On teste un cas où les formules sont équivalentes.

```

# formules_équivalentes
  (Et(Ou(Var "P1",Var "Q1"),Var "P2"))
  (Ou(Et(Var "P2",Var "P1"),Et(Var "Q1",Var "P2"))));
- : bool = true

```

On teste un cas où les formules ne sont pas équivalentes.

```

# formules_équivalentes
  (Et(Ou(Var "P1",Var "Q1"),Var "P2"))
  (Ou(Et(Var "Q1",Var "P1"),Et(Var "Q1",Var "P2"))));
- : bool = false

```

On teste un cas où des variables n'apparaissent que dans une seule des formules.

```

# formules_équivalentes
  (Implique(Et(Ou(Var "Q1",Var "P1"),Var "P2"),Et(Var "R",Non(Var "R"))))
  ((Et(Et(Ou(Var "Q1",Var "P1"),Var "P2"),Ou(Var "L",Non(Var "L"))))));
- : bool = true

```

A.4.5 assignations_vraies

On teste sur une formule satisfiable.

```

# assignations_vraies
  (Et(Est_équivalent(Var "P1",Var "Q1"),Est_équivalent(Var "P2",Var "Q2"))));
- : (string * bool) list list =
[["Q2", true); ("P2", true); ("Q1", true); ("P1", true)];
[["Q2", false); ("P2", false); ("Q1", true); ("P1", true)];
[["Q2", true); ("P2", true); ("Q1", false); ("P1", false)];
[["Q2", false); ("P2", false); ("Q1", false); ("P1", false)]]

```

On teste sur une formule non satisfiable.

```

# assignations_vraies (Et(Var "p",Non(Var "p")));
- : (string * bool) list list = []

```

A.4.6 assignation_vraie

On teste sur une formule satisfiable.

```

# assignation_vraie
  (Et(Est_équivalent(Var "P1",Var "Q1"),Est_équivalent(Var "P2",Var "Q2")));
- : (string * bool) list =
[["Q2", true); ("P2", true); ("Q1", true); ("P1", true)]

```

On teste sur une formule non satisfiable.

```

# assignation_vraie (Et(Var "p",Non(Var "p")));
Exception: Failure "assignation_vraie : la formule n'est pas satisfiable".

```

B Le code

```
(* QUESTION 1 *)

(* Definition des types *)

type tformule =
  | Valeur of bool
  | Var of string
  | Non of tformule
  | Et of tformule * tformule
  | Ou of tformule * tformule
  | Implique of tformule * tformule
  | Est_equivalent of tformule * tformule
;;

type arbreDec =
  | FeuilleDec of bool
  | Racine of string * arbreDec * arbreDec
;;

(* Pour isoler les variables d'une formule *)

(* Interface isole_var
  Type : tformule -> string list
  Argument : formule
  Precondition : aucune
  Postcondition : renvoie la liste des variable contenue dans la formule,
                  avec des doublons possibles
                  si la formule ne contient pas de variables, renvoie la liste vide
*)

let rec isole_var formule =
  match formule with
  | Valeur _ -> []
  | Var x -> [x]
  | Non f -> isole_var f
  | Et (f1,f2)
  | Ou (f1,f2)
  | Implique (f1,f2)
  | Est_equivalent (f1,f2) -> ( isole_var f1 ) @ ( isole_var f2 )
;;

(* Interface appartient
  Type : 'a -> 'a list -> bool
  Argument : element, liste
  Precondition : aucune
  Postcondition : renvoie true si l'element est dans la liste, false sinon.
*)

let rec appartient e l =
  match l with
  | [] -> false
  | t::q -> ( t = e ) || ( appartient e q )
;;

(* Interface elimine_doublons
```

```

    Type : 'a list -> 'a list
    Argument : liste
    Precondition : aucune
    Postcondition : supprime les doublons d'une liste
*)

let rec elimine_doublons l =
  match l with
  | [] -> []
  | t::q -> if appartient t q then ( elimine_doublons q )
            else t::( elimine_doublons q )
;;

(* Interface variables
   Type : tformule -> string list
   Argument : formule
   Precondition : aucune
   Postcondition : renvoie la liste des variable contenues dans la formule,
                  sans doublons.
                  si la formule ne contient pas de variables, renvoie la liste vide
*)

let variables formule = elimine_doublons (isole_var formule) ;;

(* Evaluer une formule *)

(* Interface valeur_logique
   Type : string -> string * bool list -> bool
   Argument : nom de variable, environnement
   Precondition : la variable doit etre dans l'environnement
   Postcondition : renvoie la valeur de verite de la variable dans l'environnement
   Raises : failwith "valeur_logique : la variable n'est pas dans l'environnement !"
*)

let rec valeur_logique p environnement=
  match environnement with
  | [] -> failwith "valeur_logique : la variable n'est pas dans l'environnement !"
  | (t,b)::q -> if t = p then b
                else valeur_logique p q
;;

(* Interface eval
   Type : tformule -> (string * bool) list -> bool
   Argument : formule, environnement
   Precondition : l'environnement doit englober toutes les variables de la formule
   Postcondition : renvoie la valeur de verite de la formule en donnant
                  aux variables la valeur qui leur correspond dans l'environnement
*)

let rec eval formule env =
  match formule with
  | Valeur x -> x
  | Var p -> valeur_logique p env
  | Non a -> not (eval a env)
  | Et (a,b) -> (eval a env) && (eval b env)
  | Ou (a,b) -> (eval a env) || (eval b env)
  | Implique (a,b) -> (not (eval a env)) || (eval b env)
  | Est_equivalent (a,b) -> ((eval a env) && (eval b env))

```

```

|| ((not (eval a env)) && (not (eval b env)))
;;

(* Creer l'arbre *)

(* Interface : dec_of_formule
Type : tformule -> arbreDec
Argument : formule
Precondition : aucune
Postcondition : renvoie l'arbre de decision correspondant a la formule donnee
*)

let dec_of_formule formule =
  let rec aux var_liste environnement =
    match var_liste with
    | [] -> FeuilleDec (eval formule environnement)
    | t::q -> Racine (t,
                      aux q ((t,true)::environnement),
                      aux q ((t,false)::environnement)
                     )
  in
    aux (variables formule) []
;;

(*****

(* QUESTION 2 *)

(* Definition des types *)

type bddNoeud =
  | FeuilleBdd of int * bool
  | NoeudAutre of int * string * int * int
;;

(* Rechercher une adresse *)

(* Interface adresse
Type: arbreDec -> bddNoeud list -> arbreDec list -> int
Argument : arbre recherche,
           bdd (qui contient l'adresse),
           liste d'arbres de decision
Precondition : a chaque element dans la liste du bdd,
               correspond son equivalent dans la liste des sous-arbres
               l'ordre des elements doit etre le meme dans les 2 listes,
               l'arbreDec doit etre dans la liste d'arbres de decision
Postcondition : renvoie l'adresse de l'arbre recherche
Raizes : failwith "adresse: l'arbre n'est pas dans les donnees".
         failwith "adresse: le bdd et le dec ne coincident pas".
*)

let rec adresse arbre bdd donnees =
  match bdd,donnees with
  | [] , [] -> failwith "adresse: l'arbre n'est pas dans les donnees"
  | (FeuilleBdd (numero,p))::bddReste , (FeuilleDec q)::donneesReste ->
    if FeuilleDec p = arbre then numero
    else adresse arbre bddReste donneesReste
  | (NoeudAutre (numero,nom,g,d))::bddReste , (Racine(q,g2,d2))::donneesReste ->

```

```

        if (Racine (nom,g2,d2)) = arbre then numero
        else adresse arbre bddReste donneesReste
    | _ , _ -> failwith "adresse: le bdd et le dec ne coincident pas"
;;

(* Construire le BDD *)

(* Interface construitBDD
Type : arbreDec -> bddNoeud list -> arbreDec list
                                     -> bddNoeud list * arbreDec list
Argument : arbre qu'on veut transformer,
          BDD deja cree,
          liste des sous-arbres deja ajoutees au bdd,
          compteur qui servira a adresser les bddNoeuds
Precondition : a chaque element dans la liste du bdd,
               correspond son equivalent dans la liste des sous-arbres.
               Les feuilles doivent etre deja inseree dans le BDD
               et la liste des sous-arbres.
Postcondition : renvoie le bdd correspondant a l'arbre a construire,
               et la liste des sous-arbres.
               N'insere pas les feuilles.
*)

let rec construitBDD arbreDec bdd donnees=
  match arbreDec with
  | FeuilleDec _ -> (bdd,donnees)
  | Racine (nom,arbreG,arbreD) ->
    if appartient arbreDec donnees
    then (bdd,donnees)
    else if appartient arbreG donnees
         then if appartient arbreD donnees
              then (NoeudAutre(1+(List.length bdd),nom,adresse arbreG bdd donnees ,
                              adresse arbreD bdd donnees)::bdd,
                    Racine(nom,arbreG,arbreD)::donnees)
              else let (bddD,donneesD) = construitBDD arbreD bdd donnees in
                   (NoeudAutre(1+(List.length bddD),nom,adresse arbreG bdd donnees ,
                              adresse arbreD bddD donneesD)::bddD,
                    Racine(nom,arbreG,arbreD)::donneesD)
         else if appartient arbreD donnees
              then let (bddG,donneesG) = construitBDD arbreG bdd donnees in
                   (NoeudAutre(1+(List.length bddG),nom,adresse arbreG bddG donneesG ,
                              adresse arbreD bdd donnees)::bddG,
                    Racine(nom,arbreG,arbreD)::donneesG)
              else let (bddG,donneesG) = construitBDD arbreG bdd donnees in
                   let (bddD,donneesD) = construitBDD arbreD bddG donneesG in
                   (NoeudAutre(1+(List.length bddD),nom,adresse arbreG bddD donneesD ,
                              adresse arbreD bddD donneesD)::bddD,
                    Racine(nom,arbreG,arbreD)::donneesD)
;;

(* Placer les feuilles qui seront dans le BDD *)

(* Interface appartientArbre
Type : arbreDec -> arbreDec -> bool
Argument : arbre de decision (element), arbre de decision (environnement)
Precondition : on recherche un element qui est une feuille
Postcondition : teste si l'element est dans l'environnement
*)

```

```

let rec appartientArbre element arbre =
  match arbre with
  | Racine (_,arbreG,arbreD) -> (appartientArbre element arbreG)
                               || (appartientArbre element arbreD)
  | _ -> element = arbre
;;

(* Interface depart
Type : arbreDec -> bddNoeud list * arbreDec list
Argument : arbre de decision
Precondition : aucune
Postcondition : renvoie le couple compose des bddNoeuds utilises,
                puis de la liste des arbreDec utilises.
*)

let depart dec =
  if appartientArbre (FeuilleDec (true)) dec
  then if appartientArbre (FeuilleDec (false)) dec
       then [FeuilleBdd (2, false); FeuilleBdd (1, true)],
           [FeuilleDec (false); FeuilleDec (true)]
       else [FeuilleBdd (1, true)], [FeuilleDec (true)]
  else if appartientArbre (FeuilleDec (false)) dec
       then [FeuilleBdd (2, false)], [FeuilleDec (false)]
       else [], [] (* pour exhaustivite! *)
;;

(* La fonction principale *)

(* Interface bdd_of_formule
Type : tformule -> bddNoeud list
Argument : formule
Precondition : aucune
Postcondition : renvoie le BDD correspondant a la formule
*)

let bdd_of_formule formule =
  let arbreDEC = dec_of_formule formule
  in
  let departBDD, departDEC = depart arbreDEC
  in
  fst (construitBDD arbreDEC departBDD departDEC)
;;

(*****)

(* QUESTION 3 *)

(* Simplifier une fonction *)

(* Interface simplifie
Type : bddNoeud list -> int -> int -> bddNoeud list
Argument : BDD, valeur a changer, direction
Precondition : aucune
Postcondition : renvoie le BDD en modifiant les valeurs a changer
                par la direction
                ne touche pas aux feuilles
*)

```

```

let rec simplifie bdd2 achanger direction =
  match bdd2 with
  | NoeudAutre (adresse,nom,g,d)::q ->
    if adresse = achanger then simplifie q achanger direction else
    if g = achanger
      then if d = achanger
        then simplifie q achanger direction
        else NoeudAutre (adresse,nom,direction,d)
          ::(simplifie q achanger direction)
      else if d = achanger
        then NoeudAutre (adresse,nom,g,direction)
          ::(simplifie q achanger direction)
        else NoeudAutre (adresse,nom,g,d)
          ::(simplifie q achanger direction)
  | _ -> bdd2
;;

(* Interface bddSimplePartiel
Type : bddNoeud list -> bddNoeud list -> bddNoeud list
Argument : BDD, BDD bis
Precondition : BDD = BDD bis
Postcondition : renvoie le BDD en le simplifiant partiellement
*)

let rec bddSimplePartiel bdd bdd2 =
  match bdd with
  | NoeudAutre (adresse,nom,adresseG,adresseD)::q ->
    if adresseG = adresseD
      then simplifie bdd2 adresse adresseG
      else bddSimplePartiel q bdd2
  | _ -> bdd2
    (* Si feuille ou [], alors on a fini, on renvoie bdd2 *)
;;

(* La fonction principale *)

(* Interface bddSimple_of_bdd
Type : bddNoeud list -> bddNoeud list
Argument : BDD
Precondition : aucune
Postcondition : renvoie le BDD simplifie correspondant au BDD
*)

let rec bddSimple_of_bdd bdd =
  let bdd2 = (bddSimplePartiel bdd bdd) in
  if bdd2 = bdd then bdd else bddSimple_of_bdd bdd2
;;

(* Deux fonctions supplementaires *)

(* Interface bdd_of_dec
Type : arbreDec -> bddNoeud list
Argument : arbre de decision
Precondition : aucune
Postcondition : renvoie le BDD simplifie correspondant a l'arbre de decision
*)

```

```

let bdd_of_dec arbreDEC =
  let departBDD,departDEC = depart arbreDEC
  in
  bddSimple_of_bdd (fst (construitBDD arbreDEC departBDD departDEC))
;;

(* Interface bddSimple_of_formule
Type : tformule -> bddNoeud list
Argument : formule
Precondition : aucune
Postcondition : renvoie le BDD simplifie correspondant a la formule
*)

let bddSimple_of_formule formule =
  bddSimple_of_bdd (bdd_of_formule formule)
;;

(*****

(* QUESTION 4 *)

(* Afficher graphiquement un BDD *)

(* Interface dotBDD
Type : arbreDec -> unit
Argument : Arbre de decision
Precondition : aucune
Postcondition : renvoie un fichier graphique dot representant
le BDD qui correspond a l'arbre de decision
*)

let dotBDD arbre =
  let fichier = open_out "dotBDDfichier.dot"
  in
  let bdd = bdd_of_dec arbre
  in
  let rec enForme bdd =
    match bdd with
    | [] -> ""
    | FeuilleBdd (numero,p)::q ->
      ( (string_of_int numero) ^ " [style = bold, label=\"\"
      ^ (if p then \"V\" else \"F\") ^ \"\"];\\n\" ) ^ enForme q
    | NoeudAutre (numero,nom,g,d)::q ->
      (string_of_int numero) ^ " [ label=\"\"^nom(*^\" : \"
      ^ (string_of_int numero)*\" ^ \"\"];\\n\"
      ^ (string_of_int numero) ^ \" -> \" ^ (string_of_int g)
      ^ \" [color=red,style=dashed];\\n\"
      ^ (string_of_int numero) ^ \" -> \" ^ (string_of_int d) ^ \";\\n\"
      ^ enForme q
    in
    output_string fichier ("digraph G {\\n\" ^ (enForme bdd) ^ \"}");
    close_out fichier
  ;;

(*****

(* QUESTION 5 *)

```

```

(* Definition des types *)

type arbreDecNum =
  | FeuilleDecNum of int*bool
  | RacineNum of int*string*arbreDecNum*arbreDecNum
;;

(* Numeroter un arbre *)

(* Interface nbelements
Type : arbreDec -> int
Argument : arbre de decision
Precondition : aucune
Postcondition : renvoie le nombre de Racines et FeuilleDecs de l'arbre de decision
*)

let rec nbelements a =
  match a with
  | FeuilleDec _ -> 1
  | Racine (_,g,d) -> 1 + (nbelements g) + (nbelements d)
;;

(* Interface numerote
Type : arbreDec -> arbreDecNum
Argument : arbre de decision
Precondition : aucune
Postcondition : attribue une adresse distincte a chaque racine ou feuille
de l'arbre de decision
*)

let numerote arbredec =
  let rec aux arbre compteur =
    match arbre with
    | FeuilleDec p -> FeuilleDecNum (compteur,p)
    | Racine (p,g,d) -> RacineNum (compteur,p,aux g (compteur+1),
                                   aux d (compteur+1+(nbelements g)))
  in
  aux arbredec 1
;;

(* Afficher graphiquement un DEC *)

(* Interface dotDEC
Type : arbreDec -> unit
Argument : Arbre de decision
Precondition : aucune
Postcondition : renvoie un fichier graphique dot representant
l'arbre de decision
*)

let dotDEC arbredec =
  let fichier = open_out "dotDECfichier.dot"
  in
  let rec enForme arbredecnum =
    match arbredecnum with
    | FeuilleDecNum (numero,p) -> (string_of_int numero)

```

```

      ^ " [style = bold, label=\\""
      ^ (if p then "V" else "F") ^ "\\";\n"
    | RacineNum (numero,nom,FeuilleDecNum (numG,p),FeuilleDecNum (numD,q)) ->
      (string_of_int numero)^" [ label=\\""^nom^"\\";\n"
      ^ (string_of_int numero)^" -> \"^(string_of_int numG)
      ^" [color=red,style=dashed];\n"
      ^ (string_of_int numero)^" -> \"^(string_of_int numD)^\";\n"
      ^ enForme (FeuilleDecNum (numG,p))
      ^ enForme (FeuilleDecNum (numD,q))
    | RacineNum (numero,nom,RacineNum (numG,g,gg,gd),RacineNum (numD,d,dg,dd)) ->
      (string_of_int numero)^" [ label=\\""^nom^"\\";\n"
      ^ (string_of_int numero)^" -> \"^(string_of_int numG)
      ^" [color=red,style=dashed];\n"
      ^ (string_of_int numero)^" -> \"^(string_of_int numD)^\";\n"
      ^ enForme (RacineNum (numG,g,gg,gd))
      ^ enForme (RacineNum (numD,d,dg,dd))
    | _ -> failwith "DotDec : Pour exhaustivite"
  in
    output_string fichier ("digraph G {" ^ (enForme (numerote arbredec)) ^ "}");
    close_out fichier
;;

(*****

(* QUESTION 6 *)

(* Tester si la formule est une tautologie *)

(* Interface tautologie
   Type : tformule -> bool
   Argument : formule
   Precondition : aucune
   Postcondition : teste si la formule est une tautologie
*)

let tautologie form =
  bddSimple_of_formule form = [FeuilleBdd(1,true)]
;;

(*****

(* QUESTION 7 *)

(* Construire deux arbres DEC en meme temps *)

(* Interface doubleDEC
   Type : tformule -> tformule -> arbreDec * arbreDec
   Argument : formule1, formule2
   Precondition : aucune
   Postcondition : renvoie les arbres de decision correspondant
                   aux formules donnees,
                   en utilisant la meme liste de variables
*)

let doubleDEC form1 form2 =
  let rec aux form var_liste environnement =
    match var_liste with

```

```

| [] -> FeuilleDec (eval form environnement)
| t::q -> Racine (t,
                 aux form q ((t,true)::environnement),
                 aux form q ((t,false)::environnement)
                )
in
  let liste_variabels = elimine_doublons ((variables form1)@(variables form2))
  in
    ( aux form1 (liste_variabels) [] , aux form2 (liste_variabels) [] )
;;

(* Uniformiser les adresses *)

(* Interface remplace
Type : bddNoeud list -> int -> int -> int -> int -> bddNoeud list
Argument : BDD dont on ne doit pas modifier la taille
           valeur de l'adresse a changer
           valeur par laquelle on doit remplacer l'adresse
           nouvelle adresse du successeur gauche
           nouvelle adresse du successeur droit
Precondition : aucune
Postcondition : renvoie le BDD en modifiant toutes les occurences
                de l'adresse a changer par la nouvelle adresse.
                Et dans le noeud dont on a change l'adresse,
                on remplace les successeurs gauches et droits
                par les nouvelles adresses des successeurs.
                ne modifie pas les feuilles.
*)

let rec remplace bdd2fixe achanger direction nouveauG nouveauD =
  match bdd2fixe with
  | NoeudAutre (adresse,nom,g,d)::q ->
    if adresse = achanger
    then (NoeudAutre (direction,nom,nouveauG,nouveauD))
         ::remplace q achanger direction nouveauG nouveauD
    else
    if g = achanger
    then if d = achanger
         then remplace q achanger direction nouveauG nouveauD
         else NoeudAutre (adresse,nom,direction,d)
              ::(remplace q achanger direction nouveauG nouveauD)
    else if d = achanger
         then NoeudAutre (adresse,nom,g,direction)
              ::(remplace q achanger direction nouveauG nouveauD)
         else NoeudAutre (adresse,nom,g,d)
              ::(remplace q achanger direction nouveauG nouveauD)
  | _ -> bdd2fixe
;;

(* Tester si deux formules sont equivalentes *)

(* Interface formules_equivalentes
Type : tformule -> tformule -> bool
Argument : formule1, formule2
Precondition : aucune
Postcondition : teste si les formules sont equivalentes
*)

let formules_equivalentes form1 form2 =

```

```

let rec aux bdd1 bdd2 bdd2fixe securite=
  match bdd1,bdd2 with
  | NoeudAutre (adresse1,nom1,adresseG1,adresseD1)::q1,
    NoeudAutre (adresse2,nom2,adresseG2,adresseD2)::q2 ->
    if nom1=nom2 then
      let nouveauBDD1,nouveauBDD2fixe =
          aux q1 q2 (remplace bdd2fixe (adresse2) (adresse1+securite)
                        (adresseG1+securite) (adresseD1+securite))
              securite
          in
            (NoeudAutre(adresse1+securite,nom1,adresseG1+securite,
                        adresseD1+securite)::nouveauBDD1,
              nouveauBDD2fixe)
          else (bdd1,bdd2fixe)
      | _ -> (bdd1,bdd2fixe)
  in
    let arbredec1,arbredec2 = doubleDEC form1 form2
    in
      let bdd1,bdd2 = (bdd_of_dec arbredec1,bdd_of_dec arbredec2)
      in
        let securite = List.length bdd2
        in
          let (nouveauBDD1,nouveauBDD2) = aux (bdd1) (bdd2) (bdd2) (securite)
          in nouveauBDD1 = nouveauBDD2
    ;;

(*****)

(* QUESTION 8 *)

(* Chercher les assignations rendant vraie une formule *)

(* Interface assignations_vraies
Type : tformule -> (string * bool) list list
Argument : formule
Precondition : aucune
Postcondition : renvoie les assignations assignation rendant la formule vraie,
                ou la liste vide s'il n'en existe pas
*)

let assignations_vraies formule =
  let rec aux var_liste environnement =
    match var_liste with
    | [] -> if (eval formule environnement) then [environnement] else []
    | t::q -> (aux q ((t,true)::environnement))@(aux q ((t,false)::environnement))
  in
    aux (variables formule) []
  ;;

(* Interface assignation_vraie
Type : tformule -> (string * bool) list
Argument : formule
Precondition : la formule doit etre satisfiable
Postcondition : renvoie une assignation assignation rendant la formule vraie
Raises : failwith "assignation_vraie : la formule n'est pas satisfiable"
*)

let assignation_vraie formule =

```

```
match (assignments_vraies formule) with
| [] -> failwith "assignation_vraie : la formule n'est pas satisfiable"
| t::q -> t
;;

(*****
(* FIN *)
```