ECOLE NATIONALE SUPÉRIEUR D'INFORMATIQUE POUR L'INDUSTRIE ET L'ENTREPRISE

Rapport du Projet : IPF 2019-2020

Akarioh Samir

30 juin 2020

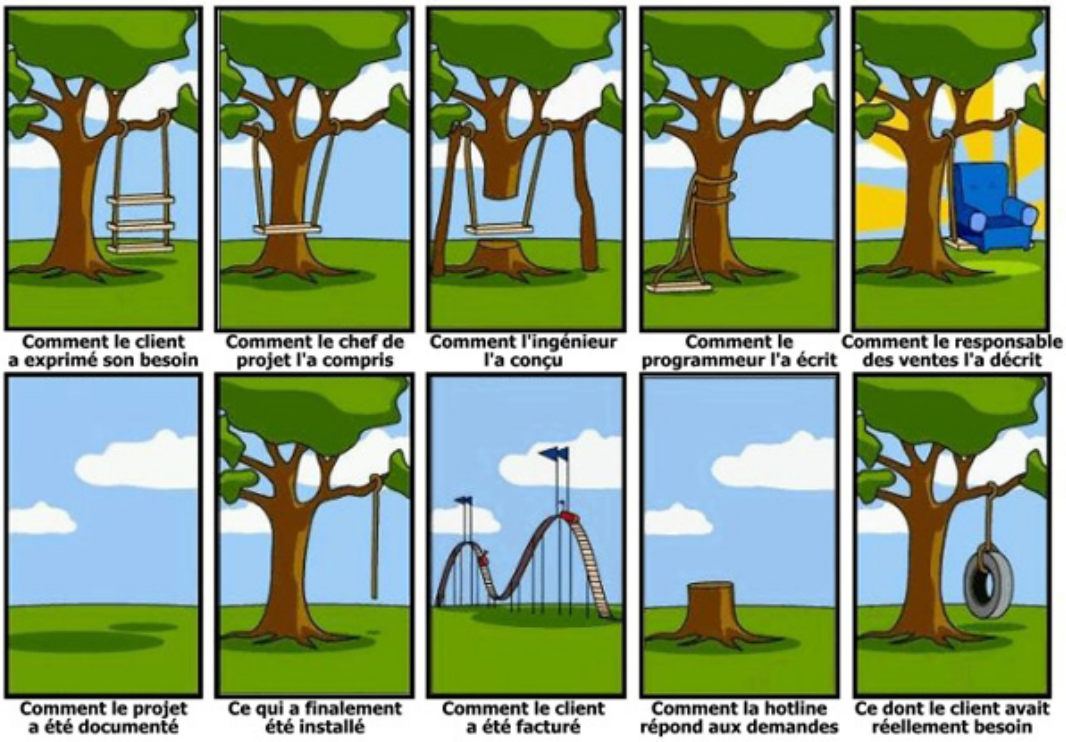

FIGURE 0.1 – Il faut avoir un cahier des charges

Important :

Avant de lancer les deux programmes,vous aurez besoin d'avoir ocamlc sur votre ordinateur.

TABLE DES MATIÈRES

1 Types utilisés **4**
1.1 Types données par le sujet 4
1.2 Type utilisée pour faciliter le projet 5

2 Fonction **5**
2.1 Question 1 5
2.1.1 Fonction annexe 5
2.1.2 Fonction Principale 6
2.2 Question 2 6
2.2.1 Fonction annexe 6
2.2.2 Fonction Principale 7
2.3 Question 3 8
2.3.1 Fonction annexe 8

2.3.2 Fonction Principale 8
2.4 Question 4 9
2.4.1 Fonction annexe 9
2.4.2 Fonction Principale 10
2.5 Question 5 10
2.5.1 Fonction annexe 10
2.5.2 Fonction Principale 11
2.6 Question 6 12
2.6.1 Fonction Principale 12
2.7 Question 7 12
2.7.1 Fonction annexe 12
2.7.2 Fonction Principale 13
2.8 Question 8 13
2.8.1 Fonction annexe 13
2.8.2 Fonction Principale 14
2.9 Question 9 15
2.9.1 Fonction annexe 15
2.9.2 Fonction Principale 16
2.10 Test 16

1 Types utilisés

1.1 Types données par le sujet

Les deux definitions de types suivante sont la modélisation des formules logiques et de leurs arbres de décision associées :

— Tformula :

```
type tformula =
  | Value of bool
  | Var of string
  | Not of tformula
  | And of tformula * tformula
  | Or of tformula * tformula
  | Implies of tformula * tformula
  | Equivalent of tformula * tformula
```

— Dectree :

```
type decTree = DecLeaf of bool | DecRoot of string * decTree * decTree
```

— env :

```
type env = (string * bool) list
```

Env est une liste de tuple dont chaque tuple est composée du nom de la variable et aussi de sa valeur logique associe c.-à-d. Vrai ou Faux cela va nous permettre par la suite d'évaluer assez facilement nos fonctions logiques .

— env :

```
type env = (string * bool) list
```

Env est une liste de tuple dont chaque tuple est composée du nom de la variable et aussi de sa valeur logique associe c.-à-d. Vrai ou Faux cela va nous permettre par la suite d'évaluer assez facilement nos fonctions logiques .

— bdd :

```
type bddNode = BddLeaf of int * bool | BddNode of int * string * int * int
type bdd = int * bddNode list
```

Ce type va nous permettre d'éliminer les doublons présent dans les arbres de décision créé par le type dectree .

1.2 Type utilisée pour faciliter le projet

— newbdd :

```
type newbdd= int * bdd
```

J'ai créé ce type afin de mieux gérer les index des différents nœuds du graphique car cela était plus simple pour moi de le faire comme cela .

— int option :

```
int option=Some(numero)|None
```

J'ai créé ce type afin de mieux gérer les fonctions indexNode et indexLeaf.

2 Fonction

2.1 Question 1

2.1.1 Fonction annexe

La fonction suivante me permet d'éliminer les doublons qui sont présent dans ma liste de variables de type str pour cela on a 4 cas l est une liste triée via liste sort,acc la liste sans doublons triée :

-Si l est vide -> renvoie la liste acc dans l'ordre inverse

-si on a un élément dans l et acc vide -> on ajoute l'élément dans acc et on continue l'appel sur le reste de l

- De même si l et acc ne sont pas vide que la tête de l est différente de acc.

- Sinon on ne rajoute rien dans acc et on continue l'appel sur le reste de la liste l sans cet élément

```
val elimine : string list -> string list -> string list
```

```
let rec elimine (l: string list) (acc: string list) : (string list) =
```

```
2    match l with
3    |[] -> List.rev acc
4    |t::q when acc=[] -> elimine q (t::acc)
5    |t::q when (t=List.hd acc)=false -> elimine q (t::acc)
6    |_::q-> elimine q acc ;;
```

2.1.2 Fonction Principale

La fonction getVars prend une formule logique de type P1=>P2 par exemple et le but de la fonction est de retourner [P1,P2] l'ordre est assuré par la fonction List.sort et il n'y a pas de doublon grâce a la fonction élimine.

Pour cela on va matcher la formule valeur par rapport aux différentes valeurs disponible sur le type et si on arrive a une valeur Var var alors on l'ajoute dans la liste et on continue cela sur le reste des sous formules de la formule par exemple la formule AND(p1,p2) est composée de là sous formule P1 et P2 :

```
1  val getVars : tformula -> string list
```

```
1  let rec getVars (valeur:tformula) : (string list) =
2
3    elimine (List.sort compare (match valeur with
4    | Value _ -> []
5    | Var var -> [var]
6    | Not formula -> getVars formula
7    | And (formula1,formula2) -> (getVars formula1)@(getVars formula2)
8    | Or (formula1,formula2) -> (getVars formula1)@(getVars formula2)
9    | Implies (formula1,formula2) -> (getVars formula1)@(getVars formula2)
10   | Equivalent (formula1,formula2) -> (getVars formula1)@(getVars formula2) )) [] ;;
```

2.2 Question 2

2.2.1 Fonction annexe

La fonction suivante me permet d'accéder a la valeur vraie ou fausse de la variable disponible dans la liste de type env elle prend comme argument la liste de type env et le nom de la var a chercher :

-Si la liste est vide -> renvoie une erreur "Variable pas trouve"

-Si on a un élément dans la liste (v,b) et que v est egale a notre var alors on renvoie b qui est la valeur vraie ou faux de notre variable

- Sinon on s'appelle récursivement sur le reste de la liste

```
val find : env -> string -> bool
```

```
let rec find (listVars:env) (var:string) : (bool) =
  match listVars with
  | []-> failwith "Variable pas trouve"
  | (v,b)::q when var=v ->  b
  | _::q -> find q var ;;
```

2.2.2 Fonction Principale

La fonction EvalFormula prend une listvar de type env et une fonction logique de type P1=>P2 par exemple et le but de la fonction est de retourner la valeur logique de la formule en fonction des valeurs logique des variables :

```
val evalFormula : env -> tformula -> bool
```

```
let rec evalFormula (listvar:env) (formula :tformula): (bool)=
    match formula with
    | Value value -> value
    | Var var -> find listvar var
    | Not formula -> not(evalFormula listvar formula)
    | And (formula1,formula2) -> evalFormula listvar formula1 && evalFormula listvar
     formula2
    | Or (formula1,formula2) -> evalFormula listvar formula1 || evalFormula listvar
    formula2
    | Implies (formula1,formula2) -> not (evalFormula listvar formula1) ||  (
    evalFormula listvar formula2)
    | Equivalent (formula1,formula2) ->((evalFormula listvar formula1) && (
    evalFormula listvar formula2)) || (not(evalFormula listvar formula1) && not(
    evalFormula listvar formula2));;
```

2.3 QUESTION 3

2.3.1 FONCTION ANNEXE

La fonction suivante me permet de créer mon arbre de décision qui prend une liste de variable s un environnement qui est créé petit a petit dans la fonction mais aussi une tformula : -Si la liste de variable est vide -> on créé un DecLeaf et sa valeur est donne par evalFormula envi formule.

-Si on créé un DecRoot suivant : DecRoot(t,creation q ((t,false) : :envi) formule,creation q ((t,true) : :envi) formule)

t est le nom de la variable,et les deux appels récursif permette de créer la suite de l'arbre pour le reste de la liste l appeller q et petit a petit on créé notre environnement en lui rajoutant vrai ou faux a chaque appel récursif .

```
val creation : string list -> env -> tformula -> decTree
```

```

let rec creation (listevar:string list) (envi:env) (formule:tformula): (decTree)  =
  match listevar with
  | [] -> DecLeaf(evalFormula envi formule)
  | t::q ->DecRoot(t,
                   creation q ((t,false)::envi) formule,
                   creation q ((t,true)::envi) formule);;
```

2.3.2 FONCTION PRINCIPALE

La fonction buildDecTree prend une formule en paramètre et nous renvoie le decTree en utilisant juste la fonction annexe :

```
val buildDecTree : tformula -> decTree
```

```
let rec buildDecTree (formule:tformula) : (decTree) =

  creation (getVars formule) [] formule;;
```

2.4 QUESTION 4

2.4.1 FONCTION ANNEXE

La fonction indexnode me permet savoir si le nœud bbdnode est déjà présent dans la bdd si oui alors on retourne Some(numéro du nœud) sinon on retourne None. De même pour indexLeaf sauf qu'on l'utilise sur les decleafs.

```
val indexNode : string -> int -> int -> bdd -> int option
val indexLeaf : bool -> bdd -> int option
```

```

let rec indexNode (varname:string) (numg:int) (numd:int) ((index,listenoeud) : bdd)
    : (int option ) =
  match listenoeud with
  | [] -> None
  | BddNode(numero,t,numerog,numerod)::q when (numerog=numg && numd=numerod &&
    varname=t) -> Some(numero)
  | _::q -> indexNode varname numg numd ((index,q) ) ;;

let rec indexLeaf (booleann:bool) ((index,listenoeud) : bdd) : (int option ) =
  match listenoeud with
  | [] -> None
  | BddLeaf(numero,booleanval)::q when (booleanval=booleann) -> Some(numero)
  | _::q -> indexLeaf booleann  (index,q)  ;;
```

La fonction buildBddAux prend en paramètre une formule une liste de variables,une bdd et un env.

On match sur la liste de var : -Si elle est vide on essaye d'insérer le decleaf qui correspond a 'evalformula de notre env et son index vaut index de la bdd+1

-Sinon on créé deux int*bdd pour chaque var une avec l'env a faux et l'autre a vrai par la suite on utilise les deux index i1 et i2 afin d'insérer les nœuds si on peut le faire les index i1 et i2 varie dû au fait que on change le numéro de l'index dans les deux cas ce qui permet de bien naviguer dans la bdd et d'avoir tout les nœuds

```
val buildBddAux : tformula -> string list -> env -> bdd -> int * bdd
```

```

```

```
2  let rec buildBddAux (formula : tformula) (vars: string list) (env: env) ( (index,
     lnoeud) as mybdd : bdd) : (int * bdd) =
3
4     match vars with
5
6       | [] -> (match indexLeaf (evalFormula env formula)  mybdd with
7               |Some(numero) -> (numero,mybdd)
8               |_ -> let newindex=index+1 in (newindex,(newindex,BddLeaf(newindex,
     evalFormula env formula)::lnoeud))
9               )
10
11      | var::suite-> let (i1, (index1,noeudg2)) = buildBddAux formula suite ((var,
     false)::env) mybdd in
12                     let (i2, (index2,noeudg3)) = buildBddAux formula suite ((var,
     true)::env)  (index1,noeudg2) in
13
14                     match indexNode var i1 i2 (index1,noeudg2) with
15                             |Some(numero) -> (numero,(index2,noeudg3))
16                             |_ -> let a=index2+1 in (a,(a,BddNode(a,var,i1,i2)::
     noeudg3 )) ;;
```

2.4.2 Fonction Principale

La fonction buildBdd prend une formule en paramètre et nous renvoie a bdd en utilisant juste la fonction annexe :

```
1  val buildBdd : tformula -> bdd
```

```
1
2   let buildBdd ( formule : tformula) : (bdd) =
3       let (indexd,(index,node))= buildBddAux formule (getVars formule ) [] (0,[]) in
4       (index,node);;
```

2.5 Question 5

2.5.1 Fonction annexe

La fonction suivante me permet de faire cela :

"On peut encore optimiser la structure précédente en éliminant les nœuds de la forme bddNode (n,s, p, p) : les successeurs gauche et droit sont alors les mêmes. L'idée est alors

d'ignorer le nœud n. Tout nœud qui a pour successeur (gauche ou droit) le nœud de numéro n aura désormais pour successeur le nœud de numéro p. La représentation graphique est a la figure suivante (Provenant du document projetIPF)" :

On supprime donc le nœuds n et tout les autres nœud ayant comme successeur gauche ou droit n alors ce n sera remplacé par p .

```
val modif : bddNode list -> int -> int -> bddNode list -> bddNode list
```

```
let rec modif (node:bddNode list) (n:int) (p:int) (acc:bddNode list) : (bddNode list
    ) =
  match node with
  |[] -> acc
  |BddNode(num, var ,numg,numd)::q when numg=n-> modif q n p (BddNode(num, var ,p,numd)
    ::acc)
  |BddNode(num, var ,numg,numd)::q when numd=n-> modif q n p (BddNode(num, var ,numg,p)
    ::acc)
  |BddNode(num, var ,numg,numd)::q -> modif q n p (BddNode(num, var ,numg,numd)::acc)
  |BddLeaf (num,boolean)::q-> modif q n p (BddLeaf (num,boolean)::acc)
```

2.5.2 Fonction Principale

La fonction SimplifyBDD permet donc de faire ce que j'ai décrit au dessus on ne lance la fonction modif que si on a le successeur de gauche égal a celui de droite sinon on s'appelle récursivement .

```
val simplifyBDD : bdd -> bddNode list -> bdd
```

```
let rec simplifyBDD ((index,noeud):bdd) (aux:bddNode list) : (bdd) =
  match noeud with
  | [] -> (index,aux)
  | BddNode (num, var ,numg,numd)::q when (numg=numd)-> simplifyBDD ((index,q)) (modif
     aux num numg [])
  | BddNode (num, var ,numg,numd) as noeud::q->  simplifyBDD ((index,q)) (noeud::aux)
  | BddLeaf (num,boolean) as leaf::q  -> simplifyBDD ((index,q)) (leaf::aux);;
```

2.6 Question 6

2.6.1 Fonction Principale

La fonction isTautology prend en paramètre une tformula on va regarder si le simplifyBDD de cette formule se résume a la liste de nœud suivant [BddLeaf(_,true)] si oui alors on renvoie vrai sinon on renvoie faux .

```
val isTautology : tformula -> bool
```

```
let isTautology (formule:tformula) : (bool) =
  let (index,noeud)= simplifyBDD (buildBdd formule ) [] in
  match noeud with
  |[BddLeaf(_,true)] -> true
  | _ -> false ;;
```

2.7 Question 7

2.7.1 Fonction annexe

La fonction suivante me permet de faire cela :

On va comparer les deux listes de nœud et si on a les mêmes nœuds a a la numérotation près alors on renvoie vrai sinon on renvoie faux plusieurs cas pour faux (la taille des listes est pas la même ou on a trop de Bddleaf et pas assez de BddNode.)

```
val comparenode : bddNode list -> bddNode list -> bool
```

```
let rec comparenode (n1:bddNode list ) (n2:bddNode list ) : (bool) =
  match (n1,n2) with
        |[],[] -> true
        |BddNode(a,b,c,d)::q,BddNode(e,f,g,h)::r when (a=e  && c=g && d=h) ->
    comparenode q r
        |BddLeaf(a,b)::q,BddLeaf(c,d)::r when (a=c && b=d) -> comparenode q r
        |_,_-> false ;;
```

2.7.2 Fonction Principale

La fonction areEquivalent permet de savoir si les deux formules sont équivalentes pour cela on créé les deux BDD simplifie associe aux deux formules et ensuite on renvoie la valeur logique associe de comparenode listenoeud1 listenoeud2 . On peut use utiliser la fonction isequivalent mais j'ai préféré respecter l'algorithme du sujet

```
val areEquivalent : tformula -> tformula -> bool
```

```
let areEquivalent (formule1:tformula) (formule2:tformula) : (bool) =
  let (i1,n1) =  simplifyBDD (buildBdd formule1) [] in
  let (i2,n2) =  simplifyBDD (buildBdd formule2) [] in

  comparenode n1 n2 ;;
```

2.8 Question 8

2.8.1 Fonction annexe

La fonction makestrbdd me permet d'avoir le str suivant par exemple :

```
 0 [ label =" P1 "];
 0 -> 1 [ color = red , style = dashed ];
 0 -> 16 ;
```

```
val makestrbdd : int -> string -> int -> int -> string
```

```
let makestrbdd a b c d =
    (string_of_int a)
    ^ " [label=\""^b^"\"];\n"
    ^(string_of_int a) ^ " -> "
    ^(string_of_int c)^
    " [color=red,style=dashed];\n"
    ^(string_of_int a)^" -> "
```

```
8       ^(string_of_int d)^";\n"
```

La fonction streleafbdd me permet d'avoir le str suivant par exemple :

```
1   4 [ style = bold , label =" true "];
```

```
1  val streleafbdd : int -> bool -> string
```

```
1    let streleafbdd a b =
2      (string_of_int a) ^
3      " [style =bold,label=\""
4      ^ (string_of_bool b)
5      ^"\"];\n";;
```

La fonction dotBDDaux me permet d'avoir l'intérieur du fichier .dot on construit le str petit a petit via les nœuds que l'on rencontre dans liste nœuds mais aussi via les deux autres fonctions cite au dessus :

```
1  val dotBDDaux : bddNode list -> string -> string
```

```
1   let rec dotBDDaux (noeud:bddNode list) (monstr:string) =
2    match noeud with
3    |[] -> monstr;
4    |BddLeaf(a,b)::q -> dotBDDaux q ((streleafbdd a b )^monstr)
5    |BddNode(a,b,c,d)::q->dotBDDaux q (monstr^(makestrbdd a b c d));;
```

2.8.2 Fonction Principale

La fonction dotBDD permet de créer le fichier name.dot name donnée par l'utilisateur pour avoir l'intérieur du fichier on utilise les fonctions auxiliaire et la fonction principale écrit le résultat de la fonction auxiliaire dans le fichier name.dot.

```
val dotBDD : string -> bdd -> unit
```

```
let dotBDD (name:string) ((index,noeud):bdd) =
  let fic2 = open_out (name^".dot") in
  let mystrfinal="digraph G { \n"^(dotBDDaux noeud "")^"}" in

  output_string fic2 mystrfinal;
  close_out fic2;;
```

2.9 QUESTION 9

2.9.1 FONCTION ANNEXE

La fonction makestr et strleaf fonctionne de même que les deux premières fonctions auxiliaires .

La fonction taille me permet de connaitre le nombre de nœuds que l'on a dans un dectree elle sera utilisé sur les decroot afin que dans le fichier .dot un élément du sous arbre gauche n'ai pas le même numéro qu'un nœud du sous arbre droit :

```
val taille : decTree -> int
```

```
let rec taille (arbre:decTree) : (int) =
  match arbre with
  |DecLeaf _ -> 1
  |DecRoot(_,g,d) -> 1 + taille g + taille d;;
```

La fonction dotDECaux me permet d'avoir l'intérieur du fichier .dot on construit le str petit a petit via les nœud que l'on rencontre dans l'arbre mais aussi via les deux autres fonctions cite au dessus cela est assez simple car on a 3 que 3 cas possible :

-Decleaf -Decroot composée de decleaf -Decroot composée de decroot

```
val dotDECaux : decTree -> int -> string
```

```
let rec dotDECaux arbre index =
  match arbre with
  | DecLeaf(p) -> streleaf index p
  | DecRoot(nom,DecLeaf(p),DecLeaf (q)) -> let numd=index+1 in
  (makestr index nom 1 numd) ^ " ;\n "
  ^ streleaf numd p
  ^ streleaf (numd+1) q

  | DecRoot(nom,DecRoot(nomg,rootg,rootd),DecRoot(nomd,rootg1,rootd1)) -> let numd=
    index+1 in let numg=taille(DecRoot(nomg,rootg,rootd)) in
  (makestr index nom numg numd)^ " ;\n "
  ^ dotDECaux (DecRoot(nomg,rootg,rootd)) numd
  ^ dotDECaux (DecRoot(nomd,rootg1,rootd1)) (numd+numg)

  | _ -> "" ;;
```

2.9.2 Fonction Principale

La fonction dotDEC permet de créer le fichier name.dot name donnée par l'utilisateur pour avoir l'intérieur du fichier on utilise les fonctions auxiliaire et la fonction principale écrit le résultat de la fonction auxiliaire dans le fichier name.dot.

```
val dotDEC : decTree -> string -> unit
```

```
let dotDEC arbre name =
  let fic2 = open_out (name^".dot") in
  let mystrfinal="digraph G { \n"^(dotDECaux arbre 0)^"}" in

  output_string fic2 mystrfinal;
  close_out fic2;;
```

2.10 Test

Voici mes test pour ce projet les assert ont tous fonctionne donc pour les asserts la valeur retourne est celle après le egal

```
 assert(areEquivalent (Not(Not(p1))) p1 = true);
```

```
2       assert(areEquivalent (Not(And(p1, p2))) (Or(Not(p1), Not(p2))) = true);
3       assert(areEquivalent (Not(Or(p1, p2))) (And(Not(p1), Not(p2))) = true);
4       assert(areEquivalent (Or(p1, p2)) (Or(p2,p1)) = true);
5       assert(areEquivalent (And(p1, p2)) (And(p2,p1)) = true);
6       assert(areEquivalent (And(p1,Or(p2,q1))) (Or(And(p1,p2), And(p1,q1))) = true);
7       assert(areEquivalent (Or(p1,And(p2,q1))) (And(Or(p1,p2), Or(p1,q1))) = true);
8       assert(isTautology (Value true) = true);
9       assert(isTautology (Not(Value true)) = false);
10
11      print_list  (simplifyBDD (buildBdd ex1) []);
12      print_list   (buildBdd ex1)  ;
13      assert(areEquivalent (Implies(p1, p2)) (Or(Not(p1), p2)) = true);
14      assert(areEquivalent f1 f5= true);
15      assert(areEquivalent ex1 ex2 = false);
16      assert(areEquivalent ex2 ex2 = true);
17      assert(areEquivalent (Value(true)) (Value(false)) = false);
18      assert(areEquivalent (Equivalent(p1,q1)) (Equivalent(Not(p1),Not(q1))) = true);
19      assert(getVars ex1 = ["P1";"P2"; "Q1"; "Q2"]);
20      assert(evalFormula ["P1",false] ex2=true);
21      assert(evalFormula ["P1",false;"P2",false;"Q1",false;"Q2",false] ex1=true);
22      assert(buildDecTree ex1=DecRoot("P1",DecRoot ("P2",DecRoot("Q1",DecRoot ("Q2",
        DecLeaf true ,DecLeaf false ) ,
23      DecRoot ("Q2" , DecLeaf false , DecLeaf true )) ,
24      DecRoot ("Q1" , DecRoot ("Q2" , DecLeaf false , DecLeaf false ) ,
25      DecRoot ("Q2" , DecLeaf false , DecLeaf false ))) ,
26      DecRoot ("P2" ,
27      DecRoot ("Q1" , DecRoot ("Q2" , DecLeaf false , DecLeaf false ) ,
28      DecRoot ("Q2" , DecLeaf false , DecLeaf false )) ,
29      DecRoot ("Q1" , DecRoot ("Q2" , DecLeaf true , DecLeaf false ) ,
30      DecRoot ("Q2" , DecLeaf false , DecLeaf true ))))),
31      dotBDD "bdd" (buildBdd  ex1),
32      dotDEC (buildDecTree ex1) "monbeautest";;
```

Les deux print_list m'ont affiche ceci ce qui prouve qu'il fonctionne :

```
1   BddLeaf(1,true)  BddLeaf(2,false)  BddNode(3,Q2,1,2)  BddNode(4,Q2,2,1)  BddNode(5,
      Q1,3,4)  BddNode(8,P2,5,2)  BddNode(9,P2,2,5)  BddNode(10,P1,8,9)
2   index = 10
3   BddNode(10,P1,8,9)  BddNode(9,P2,7,5)  BddNode(8,P2,5,7)  BddNode(7,Q1,6,6)
      BddNode(6,Q2,2,2)  BddNode(5,Q1,3,4)  BddNode(4,Q2,2,1)  BddNode(3,Q2,1,2)
      BddLeaf(2,false)  BddLeaf(1,true)
4   index = 10
5
6
```

Et pour les test pour les fichiers dot nous avons cela a la fin :

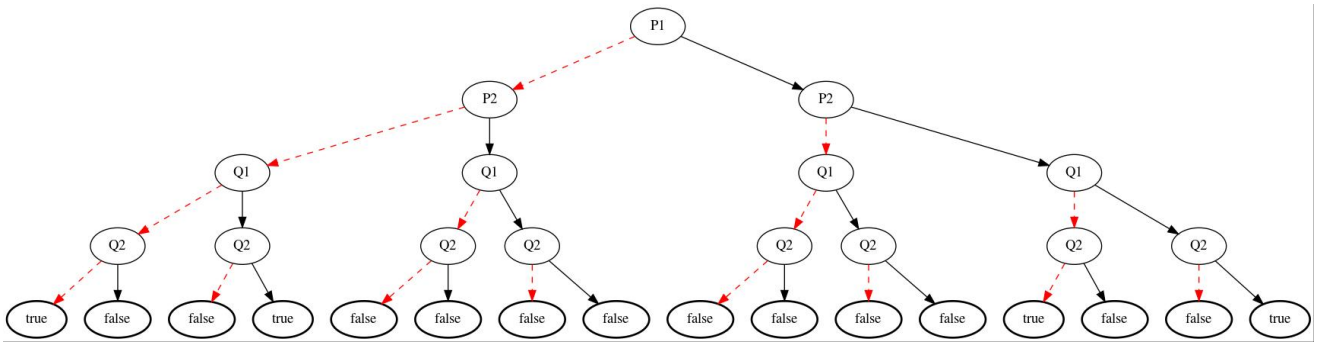

FIGURE 2.1 – Test tree

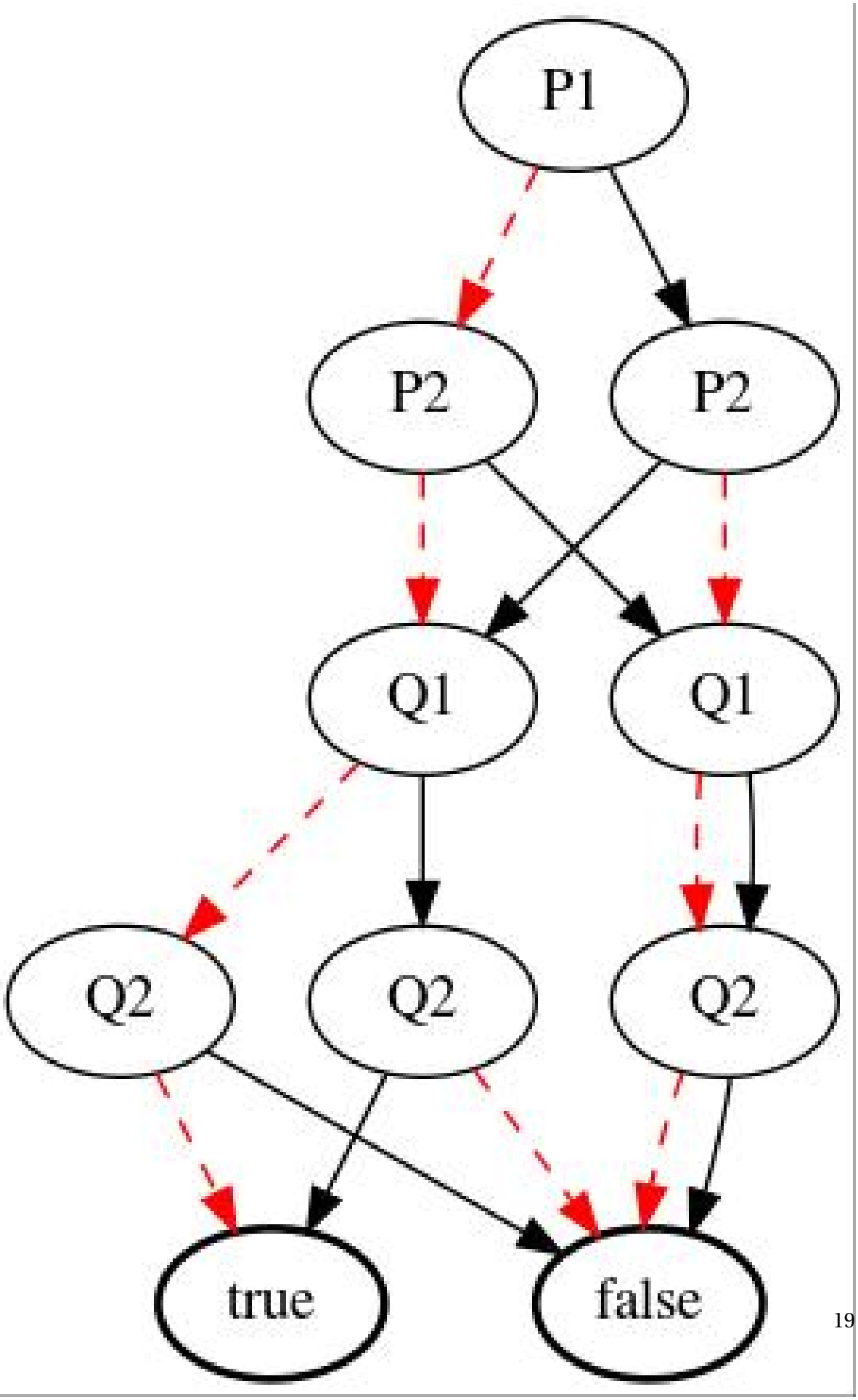

FIGURE 2.2 – Test bdd