

# Rapport Du Projet De Programmation Fonctionnelle

Loan Godard

Mai 2020

## Notice

Pour exécuter le code, il suffit de lancer le Makefile avec la commande make ou alors exécuter “ocamlc main.ml -o main.o”. ensuite il suffit de lancer l'exécutable ./main.o Pour tracer les graphes produits par le programme, il faut exécuter : dot -Tpdf -o fichier.pdf fichier.dot.

## Question 1

On définit les fonctions suivantes :

```
getVarsSorted : tformula -> string list
isIn : 'a -> 'a list -> bool
supprimeDoublon : 'a list -> 'a list
getVars : tformula -> string list
```

getVarsSorted renvoie une liste trié, contenant des doublons. C'est pour cela qu'on a créé isIn qui vérifie que l'argument est présent dans la liste et qui permet de construire récursivement supprimeDoublon qui permet de supprimer les doublons de la liste entrée. De ces fonctions on déduit getVars.

## Question 2

On définit les fonctions suivantes

```
valOfVariable : 'a -> ('a * 'b) list -> 'b
evalFormula : (string * bool) list -> tformula -> bool
```

valOfVariable permet d'obtenir la valeur définie dans l'environnement de la variable passée en argument. Par exemple si *env* = [“P1”,true;“P2”,false] alors l'appel valOfVariable “P2” *env* renvoie false. Cette fonction me permettra de construire evalFormula, une fonction récursive dans laquelle on utilise un pattern matching pour évaluer P1,P2 ...

## Question 3

On définit les fonctions suivantes

```
buildDecTree_aux : tformula -> (string * bool) list -> string list -> decTree  
buildDecTree : tformula -> decTree
```

J'ai mis du temps pour réussir cette fonction, surtout pour comprendre qu'il fallait construire la fonction récursive buildDecTree\_aux. Elle prend en argument une tformula, un *env* et une liste de variables. Si la liste de variable est vide, alors on renvoie une feuille contenant(evalFormula *env* f) et si non on renvoi un arbre contenant la tête de la liste des variables avec comme fils un appel recursif debuildDecTree\_aux avec comme environnement la concaténation de *a* avec *env* où *a*=(la tête de la liste des variables,false) pour le fils de gauche et *a*=(la tête de la liste des variables, true) pour le fils de droite.

Alors pour réaliser la fonction , il suffit d'appeler la fonction que l'on vient de construire avec l'environnement égale à une liste vide et la liste des variables égale à l'ensemble des variables trié dans l'ordre obtenue avec getVars f. Dans ce cas, on veut pour chaque chemin de l'arbre construire l'environnement jusqu'à arriver à une feuille dans laquelle on aura plus qu'à evaluer f avec l'environnement construit lors du parcours de l'arbre... Et c'est ce que fait notre fonction buildDecTree\_aux.

## Question 4

On définit les fonctions suivantes

```
isInNode : bddNode -> bddNode list -> bool  
getNextNumber : bddNode list -> int
```

getNextNumber prend une liste de bddNode et renvois le numéro du prochain noeud. Pour construire le bdd d'une formule, j'ai voulu parcourir l'arbre de décision puis ajouter le noeud dans le bdd s'il n'est pas déjà présent. (on vérifie cela avec isInNode). Cependant je ne parviens pas à bien parcourir l'arbre et je ne vois pas quoi faire quand le noeud est déjà présent... J'ai décidé de passer cette question après de nombreux essais et il fallait que j'avance sur le projet pour me redonner confiance. Toutefois, je suis revenu à cette question après avoir terminé le sujet, mais en vain.

## Question 5

On définit les fonctions suivantes

```
listeDesNoeudsASimplifier : bddNode list -> bddNode list  
isLinkIn : bddNode list -> bddNode -> int  
extractNode : int -> bddNode list -> bddNode
```

```

simplifyLOfBdd : bddNode list -> bddNode list
clearSimplifiedNode : bddNode list -> bddNode list
simplifyBdd : 'a * bddNode list -> 'a * bddNode list

```

La première fois que j'ai lu cette question, je l'ai mal interprété et je retirais de la bdd tous les noeuds de la forme  $(n,p,q,q)$ , sans modifier les autres. C'est seulement en faisant les tests à la fin du projet que je me suis rendu compte qu'elle était fausse et en faite pas si simple.

Pour construire la fonction attendu je construis d'abord listeDesNoeudsASimplifier qui prend une liste L de bddNode et renvoie une liste de bddNode contenant tous les noeuds de type  $BddNode(a,b,p,p)$  de L. Ensuite, je construis isLinkIn qui prend en argument une liste de bddNode, un bddNode et renvoie n si le bddNode entré est connecté à un bddNode de la liste et renvoie 0 sinon. Après je construit extractNode qui prend en argument un entier n correspondant au numéro d'un bddNode, une liste de bddNode et renvoie le bddNode numéroté n. Enfin je code la fonction simplifyLOfBdd qui simplifie la liste de bddNode qui compose une bdd. Cette fonction est récursive. Pour simplifier la liste entrée, on analyse tour à tour ses éléments en les extrayants. Si l'élément extrait est une feuille, on ne fait rien, si c'est un noeud, on l'analyse : s'il est connecté à un noeud présent dans la liste construite à l'aide de listeDesNoeudsASimplifier (on vérifie cela grâce à isLinkIn) alors on extrait le Noeud correspondant de cette dernière liste à l'aide de notre fonction extractNode et on connecte le noeuds que nous sommes en train d'analyser au fils du noeuds extrait. La fonction simplifyLOfBdd modifie simplement certains noeuds, sans extraire les noeuds de type  $(n,p,q,q)$  on nettoie alors notre liste à l'aide de la fonction clearSimplifiedNode qui retire de la liste entrée tous les noeuds de type  $(n,p,q,q)$ . De toutes ces fonctions, on déduit la fonction simplifyBdd.

## Question 6

On définit la fonction suivantes

```
isTautologyBdd : 'a * bddNode list -> bool
```

Dans cette question, nous avons besoin de la fonction écrite à la question 4 pour construire la bdd à partir de la tformula. Comme je n' ai pas réussi cette fonction, j' ai fais la fonction isTautologyBdd qui prend une bdd et qui renvoie un booléen : true si la bdd représente une tautologie et false sinon. Pour représenter une tautologie la Bdd ne doit contenir qu' une feuille « true ». On teste alors cela et on renvoie un booléen en conséquence. J'ai laissé en commentaire la fonction isTautology attendus à cette question : il suffit d'extraire la bdd avec buildBdd et d'utiliser la fonction isTautologyBdd.

## Question 7

On définit les fonctions suivantes

```
areEqualsBdd : 'a * 'b list -> 'a * 'b list -> bool  
areEqualsList : 'a list -> 'a list -> bool
```

Il nous faut encore la fonction buildBdd pour réaliser cette question. On la réalise alors à partir d'une Bdd. Pour tester si deux Bdd sont équivalentes, on test l'égalité entre les deux liste de Bdd noeuds avec areEqualsList. Alors si les deux listes sont égale et que le numéro du Bdd sont égaux, les Bdd sont égales. Dans ce cas, pour tester si deux Bdd sont équivalentes, on test si leurs simplification sont égales. Comme à la question précédente, je laisse la fonction areEquivalent en commentaire car on ne peux pas la tester sans la fonction buildBdd.

## Question 8

On définit les fonctions suivantes

```
string_of_bool : bool -> string  
dotDeclareNode : out_channel -> bddNode list -> bool\pounds  
dotDeclareLink : out_channel -> bddNode list -> bool  
extract_l_of_bdd : 'a * 'b -> 'b  
dotBDD : string -> 'a * bddNode list -> unit
```

Le plus long dans cette fonction a été de prendre en main l'écriture dans un fichier en ocaml. Une fois cela fait, j'ai écrit la fonction dotDeclareNode qui déclare les noeuds selon la syntaxe de dot et la fonction dotDeclareLink qui déclare les liens entre les noeuds. On fait attention a bien respecté le cahier des charges au niveau des labels et style de l'arbre attendus. string.of\_bool converti un booléen en chaîne de caractère et extract\_l\_of\_bdd extrait la liste de bddNode qui compose un bdd. dotBdd ouvre le fichier, déclare dedans un graphe G puis appelle dotDeclareNode et dotDeclareLink qui déclare le corps du graphe puis on ferme le graphe et le fichier.

## Question 9

On définit les fonctions suivants

```
dotDec_aux : decTree -> int -> out_channel -> unit  
dotDec : decTree -> string -> unit
```

Pour tracer les arbres, on remarque que les fils gauche sont numéroté par un nombre paire et les fils droit par un nombre impair. On construit alors dotDec\_aux qui prend en argument un arbre de décision, un entier  $i$  et un fichier (out\_channel) dans lequel on

vas écrire récursivement les noeuds et leurs liens. On déclare les noeuds et liens puis on rappelle la fonction avec le fils gauche et l'entier  $2i$  et un autre appelle avec le fils droit et l'entier  $2i + 1$ . Ainsi on vas parcourir l'arbre entièrement et le déclarer entièrement dans le fichier. On fait toujours attention au style et aux labels attendus. De la même manière qu'à la question précédente, pour construire dotDec on ouvre le fichier, on déclare un graph G et on appelle dotDec\_aux avec l'arbre de départ, l'entier 1 et un bien-sur le fichier. Puis on ferme le graphe et le fichier.

## Les test

Vous trouverez les tes à la fin de chaque question dans le code source. Nous testons nos fonctions avec une formule dont on connaît les résultats attendus (avec un test grâce à assert).

On vas tester nos fonctions avec la formule suivante :

$$(A1 \Leftrightarrow !A2) \vee (!B1 \Leftrightarrow B2)$$

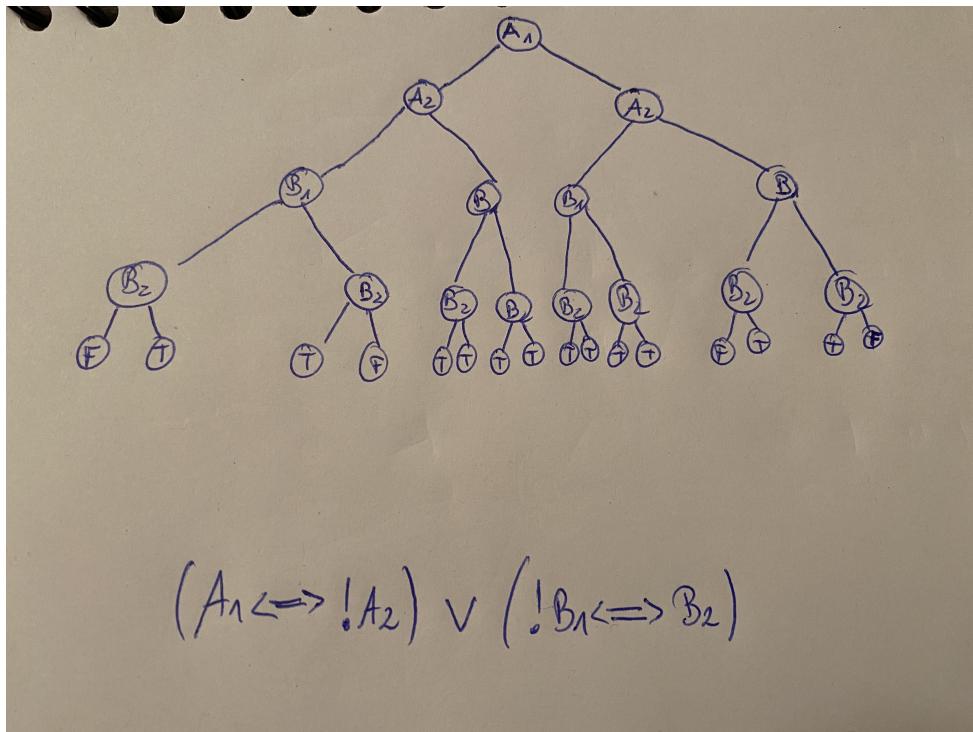
On construit les tables de vérités.

A1	A2	$A1 \Leftrightarrow !A2$
T	T	F
T	F	T
F	T	T
F	F	F

B1	B2	$!B1 \Leftrightarrow B2$
T	T	F
T	F	T
F	T	T
F	F	F

$A1 \Leftrightarrow !A2$	$!B1 \Leftrightarrow B2$	$(A1 \Leftrightarrow !A2) \vee (!B1 \Leftrightarrow B2)$
T	T	T
T	F	T
F	T	F
F	F	T

On en déduit l'arbre suivant, construit à la main :



et la bdd attendu est :

```
(9,
[BddNode (9, "A1", 7, 8); BddNode (8, "A2", 6, 4); BddNode (7, "A2", 4, 6);
BddNode (6, "B1", 5, 5); BddNode (5, "B2", 1, 1); BddNode (4, "B1", 2, 3);
BddNode (3, "B2", 1, 0); BddNode (2, "B2", 0, 1); BddLeaf (1, true);
BddLeaf (0, false)])
```

la bdd simplifié attendu est :

```
(9,
[BddNode (9, "A1", 7, 8); BddNode (8, "A2", 1, 4); BddNode (7, "A2", 4, 1);
BddNode (4, "B1", 2, 3); BddNode (3, "B2", 1, 0); BddNode (2, "B2", 0, 1);
BddLeaf (1, true); BddLeaf (0, false)])
```

Tous les test sont présents à la fin de chaque question dans le code source.

Les test que j'ai effectué se sont tous bien passé, les résultats que donne les fonctions sont bien les résultats attendus. Les graphes correspondent bien à ma fonction de départ.

## Conclusion

Pour conclure, j'ai apprécié travailler sur ce projet car il n'était pas facile mais pas non plus infaisable. J'ai passé du temps à chercher certaines questions et je regrette de ne pas avoir réussi à faire la question 4, mais je pense que parfois il faut savoir abandonner pour avancer... Le sujet était intéressant et cohérent avec l'UE. J'ai découvert la programmation fonctionnelle avec le ocaml cette année, un langage que je trouve assez particulier et intéressant.