Projet IPF - Formules logiques et BDD

Timothé Rios

01 mai 2020

Sommaire

Introduction

Le sujet de ce projet est la représentation en OCaml des formules logiques et de leurs arbres de décision. Les premières parties concernent l'implémentation des fonctions logiques ainsi que plusieurs types d'arbres de décision. les suivantes s'intéressent à la simplification de ces arbres et à deux propriétés des formules logiques. Enfin, ce rapport se termine par la réprésentation graphique des différents types d'arbre de décision.
Le code OCaml de ce projet se trouve dans le fichier projet.ml et l'interface de celui-ci dans projet.mli. Le code des tests des fonctions se trouve dans le fichier tests.ml et l'éxecutable tests.exe les affiche.

1 Types et fonctions utiles

1.1 Types

Formule logique

```
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
```

Arbre de décision

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

Environnement

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

nœud de Binary Decision Diagram

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

Binary Decision Diagram

```
type bdd = (int * (bddNode list))
```

Arbre de décision numéroté

```
type decTreeNbr =
|DecLeafNbr of int*bool
|DecRootNbr of int*string*decTreeNbr*decTreeNbr
```

1.2 Fonctions utiles

length

```
Argument : une liste l
Retourne : la longueur de l
```

isin

```
Arguments : une liste l et un élément q
Retourne : true si q est dans l, false sinon
```

unic

```
Argument : une liste l
Retourne : true si l ne contient aucun éléments en double, false sinon
```

2 Abres de décision

Le but de cette partie est de construire, à partir d'une formule logique dont les variables ont été ordonées, un arbre binaire lui correspondant, de façon à ce que chaque nœud de l'arbre corresponde à une variable et que, pour un nœud donné et sa variable, le sous-arbre gauche corresponde à la formule en ayant remplacé cette variable par false et le sous-arbre droit à la formule en ayant remplacé cette variable par true.

2.1 Ordonner les variables

La première étape est donc l'ordonnancement des variables de la formule dont on veut déterminer l'arbre de décision. La fonction `getVars t` va donc prendre en argument une formule et logique. Elle va tout d'abord utiliser une fonction auxiliaire prenant en argument une formule et une liste et dont le but est de parcourir la formule et de ranger chaque variable trouvée dans la liste. Il ne reste plus qu'à lancer cette fonction auxiliaire avec comme arguments `t` et `[]`, puis de trier la liste obtenue et d'enlever les doublons (grâce à `List.sort` et `unic`).

getVars
```
Argument : une formule t
Retourne : la liste des variables de t
```

2.2 Évaluer une formule

Après avoir obtenu la liste de ses variables, on peut évaluer une formule. Pour ce faire, on utilise le type `env` qui est une liste de couple de strings et de booléens. La fonction `findvalue env s` prend en argument un environnement `env` et un string `s`. Elle parcourt l'environnement jusqu'à trouver un couple dont le string vaut `s` et retourne le booléen associé. Si l'environnement ne contient pas `s`, `findvalue` affiche l'erreur "L'environnement ne contient pas la valeur.". Ensuite, la fonction `evalFormula env f`, qui prend en argument une formule `f` et un environnement `env` dans lequel toutes les variables de `f` sont présentes, va parcourir `f` et appliquer à chaque variable rencontrée sa valeur trouvée dans `env`, tout en appliquant les opérations logiques de `f`.

findvalue
```
Arguments : un environnement env et un string s
Retourne : le booléen du couple de env contenant s
```

evalFormula
```
Arguments : un environnement env et une fonction f
Retourne : la valeur de f dans env
```

2.3 Construire l'arbre de décision

Afin de construire l'arbre de décision, la fonction `buildDecTree f` utilise une fonction auxiliaire prenant en arguments une formule, une liste de strings ainsi qu'un environnement. Elle parcourt la liste et, à chaque élément `e` de celle-ci, construit un arbre de décision dont les deux sous-arbes sont un appel à elle-même avec comme arguments la même formule, la même liste privée de `e` et le même environnement auxquel on a rajouté le couple `(e,false)` pour le sous-arbre gauche et un appel à elle-même avec comme arguments la même formule, la même liste privée de `e` et le même environnement auxquel on a rajouté le couple `(e,true)` pour le sous-arbre droit. Arrivée à la fin de la liste, elle termine l'arbre en évaluant la formule pour créer les feuilles correspondantes grâce à `evalFormula`. Il ne reste plus qu'à appeler cette fonction auxiliaire avec comme arguments `f`, `getVars f` et `[]`.

buildDecTree
```
Argument : une fonction f
Retourne : l'arbre de décision de f
```

3 Binary Decision Diagram

Dans cette section, l'objetif est de supprimer les parties redondantes des arbres de décison. En effet, certains sous-arbres d'un arbre de décision peuvent être identiques, leur répétition n'est donc pas nécessaire. Les *Binary Decision Diagrams (BDD)* sont une nouvelle représentation des formules logiques, tirés des arbres de décision mais sans répétition inutile.

3.1 Principe de l'algorithme

Afin de construire notre Bdd, nous allons construire en parallèle une liste des arbres de décision déjà implémentés dans le Bdd, qui servira à déterminer quel sous-arbre est déjà présent dans le Bdd et donc à nous éviter de le répéter.

3.2 Initialisation

Pour construire notre Bdd et notre liste d'arbres, nous allons commencer par construire un Bdd et une liste d'arbre contenant les feuilles `true` et/ou `false`, selon notre besoin. C'est la fonction `initBdd` qui s'en chargera. Celle-ci prend en argument un arbre de décision et renvoie un Bdd et une liste d'arbre de décision contenant les feuilles présentes dans l'arbre passé en argument. Afin de détecter la présence des feuilles, nous aurons besoin de la fonction `isDecLeafIn` qui prend en arguments deux arbres de décision -le deuxième étant une feuille- et renvoie `true` si le deuxième est dans le premier.

initBdd
```
Argument : un arbre de décision dec
Retourne : un Bdd et une liste d'arbres de décision contenant les feuilles contenues dans dec
```

isDecLeafIn
```
Arguments : deux arbres de décision dec1 et dec2
Retourne : true si dec2 est dans dec1, false sinon
```

3.3 Induction

La fonction principale est `buildBddFromDecTree`, c'est elle qui construit par induction le Bdd ainsi que la liste des arbres de décision, même si elle ne renvoie que le premier. Elle prend en argument un arbre de décision qui est celui dont nous allons construire le Bdd, ainsi qu'un Bdd et une liste d'arbres de décision qui sont ceux que nous construisons à chaque étape. `buldBddFromDecTree` parcourt l'arbre de décision. Pour chaque élément de celui-ci, si l'élément est une feuille, la fonction renvoie simplement le Bdd et la liste d'arbre de décision. Si c'est un arbre, la fonction regarde si cet arbre est déjà présent dans la liste d'arbre de décision. Si c'est le cas, elle renvoie ses arguments sans les changer. Sinon, elle teste si les sous-arbres de l'élément sont dans la liste d'arbre de décision. Elle les ajoute ensuite s'il le faut à la liste et renvoie le Bdd avec un nouveau BddNode correspondant à l'élément ainsi que la liste d'arbre de décision avec l'élément en plus.
Afin d'aggrandire notre Bdd, nous avons besoin de pouvoir connaître l'adresse d'un sous-arbre. La fonction `grabDecTreeInt` prend en arguments l'arbre de décision dont on cherche l'adresse ainsi qu'un Bdd et sa liste d'arbres de décision correspondant. Elle parcourt ensuite le Bdd et la liste d'arbre de décision et, dès qu'elle trouvre l'arbre, elle renvoie l'adresse du Bdd correspondant. Si elle ne trouve pas l'arbre, `grabDecTreeInt` affiche l'erreur "arbre introuvable". Si le Bdd et la liste d'arbres ne correspondent pas, `grabDecTreeInt` affiche l'erreur "probleme de correspondance des arguments".
Enfin, il ne reste plus qu'à combiner `buildDecTree` et `buildBddFromDecTree` afin d'obtenir la fonction `buildBdd`, qui prend en argument une fonction f et retourne son Bdd correspondant.

grabDecTreeInt
```
Arguments : l'arbre dont on cherche l'adresse dec, un Bdd et une liste d'arbres de décision
Retourne : l'adresse du Bdd dont dec est le correspondant dans la liste d'arbres
```

buildBddFromDecTree
```
Arguments : un arbre de décision dec, le Bdd et la liste d'arbres de décision que l'on construit
par induction
Retourne : le Bdd et la liste d'arbres de décision correspondant à dec
```

buildBdd
```
Argument : une fonction f
Retourne : le Bdd correspondant à f
```

4 Bdd simplifié

Il est possible de simplifier encore une fois nos éléments, en supprimant les cas où les successeurs d'un nœud sont les mêmes. Il faut donc remplacer chaque lien vers un nœud dont les successeurs sont identiques par un lien vers le successeur unique, puis supprimer les nœuds qui ne sont plus utilisés dans le Bdd global.

4.1 Remplacer les liens

La première étape est simple : la fonction `splf` prend en argument une liste de nœuds, un nœud `redundant` ainsi q'un nœud `target`. Elle parcourt la liste de nœuds et, pour chaque élément de celle-ci, remplace son successeur de gauche et/ou de droite par `target` si celui-ci est `redundant`.

splf
`Arguments: une liste de nœuds, un nœud redundant et un nœud target`
`Retourne: la liste de nœuds dans laquelle les liens vers redundant ont été remplacés par des liens vers target`

4.2 Supprimer les nœuds inutiles

Ensuite, la fonction `deleteRedundant` s'occupe de supprimer d'une liste de nœuds les nœuds qui ne sont les successeurs d'aucun autre. Pour ce faire, elle utilise une fonction auxiliaire `isRedundant` prenant comme arguments une liste de nœuds et un nœud `target`. Elle parcourt la liste et renvoit `true` si aucun nœud n'a comme successeur `target`, `false` sinon. `deleteRedundant`, quant à elle, prend en arguments deux liste de nœuds identiques et parcourt la première. À chaque nœud de celle-ci, s'il n'est le successeur d'aucun autre nœud (*cf* `isRedundant`), elle le supprime de la liste.

deleteRedundant
`Arguments : deux listes de nœuds identiques`
`Retourne : la liste dont les nœuds inutiles ont été supprimés`

4.3 Simplifier le Bdd

Enfin, la fonction `simplifyBdd`, qui prend en argument un Bdd, se charge de finaliser l'alogrithme de simplification. Pour ce faire, elle utilise une fonction auxiliaire `aux` qui prend en arguments deux listes de nœuds identiques et parcourt la première. Si un de ses éléments a ses deux successeurs identiques, elle applique `splf` avec comme `redundant` l'élément et comme `target` son unique successeur. `simplifyBdd` extrait la liste de nœud du Bdd passé en argument et la passe à son tour à `aux`, en faisant attention au nœud racine ; en effet, comme `deleteRedundant` utilise pour détecter les nœuds inutiles leur absence de l'ensemble des successeur des nœuds de la liste, il supprime aussi la racine. Il faut donc tester à part si la racine est inutile (si ses successeurs sont identiques) et l'ajouter (ou non) à la fin, après avoir appliqué `deleteRedundant` à la liste de nœuds obetnue par `aux`.

simplifyBdd
`Argument : un Bdd bdd`
`Retourne : le Bdd simplifié de bdd`

5 Propriétés de formules

5.1 Tautologie

Une formule logique est une tautologie si elle est vraie quelque soit la valeur de ses variables. Déterminer si une formule est une tautologie peut se faire à l'aide des Bdd simplifiés ou sans.

Tautologie sans Bdd

Pour tester si une formule est une tautologie, il faut donc tester si elle est vraie pour toute valeur de sa liste de variables. Pour une formule `f` donnée, la première étape est donc d'avoir la liste de toutes les valeurs possibles d'un de ses environnements. Un environnement étant une liste de couple de variables de `f` et de booléens, cela revient à avoir la liste des listes de booléens de taille celle de la liste des variables de `f`. Pour ce faire, nous allons utiliser `addTrueFalse`, une fonction qui prend en argument une liste de listes de booléens, et dédouble chaque éléments de cette liste de liste en lui ajoutant `true` ou `false`. À partir de cette fonction,en l'appelant `n` fois, `boolListsSizeN n` va créer une liste de listes de booléens de taille `n`. Puis `boolListsFormula f` va pouvoir créer la listes de toutes les listes de booléens nécessaires en récupérant le nombre de variables de `f` grâce à `getVars` et `length`. Enfin, `boolListToEnv` et `envLists` vont se charger de créer la liste de tous les environnements possibles de `f`. Il ne reste plus qu'à tester la validité de `f` pour tous ses environnements, se dont se charge `isTautologyNoBdd`.

addTrueFalse
`Argument : une liste de listes l de booléens`
`Retourne : une liste contenant toutes les listes de l avec true devant et toutes les listes de l avec false devant`

boolListsSizeN
`Argument : un entier n positif`
`Retourne : la liste de toutes les listes de booléens de taille n`

boolListsFormula
`Argument : une formule f`
`Retourne : la liste de toutes les listes de booléens de taille le nombre de variables de f`

boolListToEnv
`Arguments : une formule f et une liste de booléens de taille le nombre de variables de f`
`Retourne : Un environnement de f`

envLists
`Argument : une formule f`
`Retourne : la liste de tous les environnements de f`

isTautologyNoBdd
`Argument : une formule f`
`Retourne : true si f est une tautologie, false sinon`

Tautologie avec Bdd

Si une formule est une tautologie, la liste de nœuds de son Bdd simplifié se compose uniquement de la feuille `true`. Pour tester si une formule est une tautologie, il suffit de vérifer que la longueur de la liste de nœud de son Bdd simplifié est 1.

isTautology
`Argument : une formule f`
`Retourne : true si f est une tautologie, false sinon`

5.2 Équivalence

En logique, deux propositions P et Q sont **logiquement équivalentes** si P est vraie lorsque Q est vraie et que P est fausse lorsque Q est fausse. Si P et Q sont logiquement équivalente, on note $P \equiv Q$.

Équivalence sans Bdd

Pour vérifier si deux formules `f1` et `f2` sont équivalentes, il suffit de vérifer que `f1` $\Leftrightarrow$ `f2` est une tautologie. Ainsi `AreEquivalentNoBdd f1 f2` appelle `isTautologyNoBdd (Equivalent(f1,f2))` .

areEquivalentNoBdd
`Arguments : deux formules f1 et f2`
`Retourne : true si f1` $\equiv$ `f2, false sinon`

Équivalence avec Bdd

Il est établi dans le sujet que deux formules sont équivalentes si leurs Bdd simplifiés construits avec le même ordre pour les variables sont identiques au numérotage des nœuds près. Nous allons donc construire les Bdd simplifiés des deux formules dont nous voulons tester cette nouvelle équivalence en prenant le même ordre pour les variables. Afin d'avoir le même ordre pour les variables, la fonction `buildTwoDecTrees f1 f2` va construire les arbres de décision des formules `f1` et `f2` à partir d'une même liste de variables contenant toutes celles des deux formules. Ensuite, la fonction `replaceAdd` aura pour rôle de modifier les adresses d'un des deux Bdd afin de les rendre identiques au premier. Pour cela, afin de ne pas utiliser d'adresse déjà existante, elle aura besoin de connaître la taille du Bdd, donnée par la fonction `lengthBdd`. Enfin, la fonction `areEquivalent` pourra déterminer si les deux formules passées en arguments sont équivalentes. Elle utilise pour ce faire une fonction auxiliaire qui prend en argument deux Bdd, une copie du second ainsi qu'un entier et parcourt les deux Bdd en même temps. Si, à chaque nœud rencontré, les variables contenues dedans sont identiques, elle utilise `replaceAdd` pour modifier les adresses du premier Bdd et utilise le second pour vérifier comment le modifier pour l'homogénéiser avec le premier, tout en effectuant ces modifications dans sa copie. En revanche, si les adresses ne sont pas

identiques, cela veut dire que les Bdd sont différents ; la fonction auxiliaire renvoie alors le premier Bdd ainsi que la copie du second. Il ne reste plus à `areEquivalent` de construire les Bdd simplifiés des deux formules passées en arguments, puis de passer ces derniers par `aux` avant de comparer leurs listes de nœuds pour statuer sur l'équivalence des formules.

lengthBdd
`Argument : un Bdd`
`Retourne : la longueur de sa liste de nœuds`

replaceAdd
`Arguments : une liste de nœuds et quatre entiers before, after, newG et newD`
`Retourne : la liste de nœuds dont les éléments ont été modifiés : si leur numéro était before alors on le change par after et on change ses successeurs par newG et newD. Sinon, si ses successeurs sont before on le supprime. Si un seul est before on le remplace par after`

buildTwoDecTrees
`Arguments : deux formules`
`Retourne : les arbres de décision des deux formules construits avec le même ordre de variables`

bddlist
`Argument : un Bdd`
`Retourne : sa liste de nœuds`

areEquivalent
`Arguments : deux formules f1 et f2`
`Retourne : true si f1 et f2 sont équivalentes, false sinon`

6 Affichage graphique

6.1 Affichage des Bdd

La fonction `dotBdd` s'occupe de fournir un affichage graphique aux Bdd. Pour cela, elle utilise une fonction auxiliaire `dotString` qui parcourt la liste de nœuds du Bdd passé en argument et renvoie une série de commandes interprétable par le programme `dot`. `dotBdd` écrit ensuite cette série de commande dans le fichier dont le nom a été passé en argument.

dotBdd
`Arguments : un nom de fichier nomfichier et un Bdd bdd`
`Retourne : écrit dans nomfichier la représentation graphique de bdd interprétable par dot`

6.2 Affichage des arbres de décision

L'affichage des arbres de décision est un peu plus compliqué que celui des Bdd, car, pour écrire les commandes des fichhiers DOT, nous avons besoin d'éléments numérotés, ce qui est le cas des nœuds des Bdd mais pas de ceux des arbres de décision. C'est pourquoi il faut introduire le type `DecTreeNbr`, qui ne change du type `DecTree` que par l'entier attaché à chaque nœud et feuille.La fonction `nbrTree dectree` s'occupe de compter les nœuds et feuilles de dectree et est utilisée par la fonction `addNbrTree dectree` qui se charge d'affecter à tous les nœuds et feuilles de `dectree` un entier singulier. Enfin, la fonction `dotDecTree` s'occupe du même travail que `dotBdd` mais en partant d'un arbre de décision.

nbrTree
`Argument : un arbre de décision dectree`
`Retourne : le nombre de feuilles et nœuds de dectree`

addNbrTree
`Argument : un arbre de décision dectree`
`Retourne : l'arbre de décision numéroté équivalent à dectree`

dotDecTree
`Arguments : un nom de fichier nomfichier et un arbre de décision dectree`
`Retourne : écrit dans nomfichier la représentation graphique de dectree interprétable par dot`

6.3 Exemples d'arbre de décision et de Bdd

On utilise comme exemple la formule `f` $= \Big(\neg A \vee \big(D \Rightarrow (A \wedge B)\big)\Big) \wedge \Big(\neg D \wedge \big((D \wedge C) \vee B\big) \Leftrightarrow \big(A \Rightarrow C\big)\Big)$

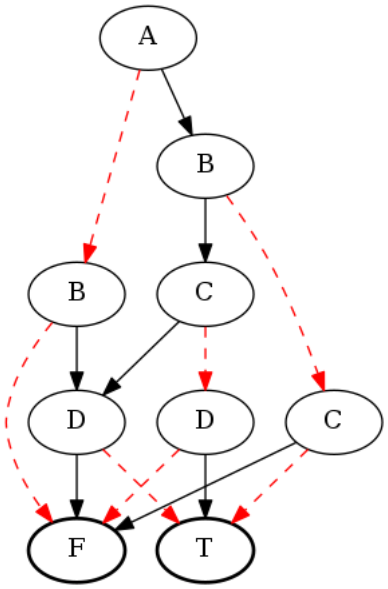

FIGURE 1 – graphe du Bdd de `f`

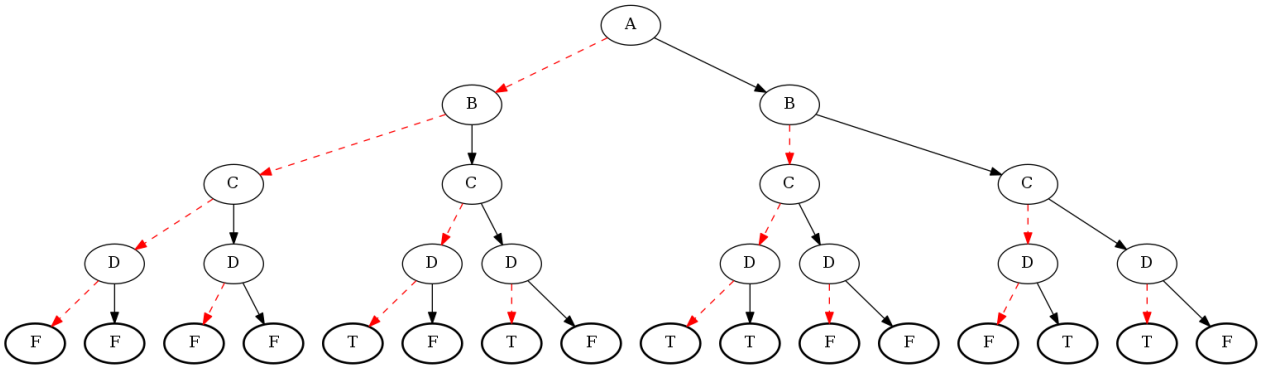

FIGURE 2 – graphe de l'arbre de décision de `f`

A Récapitulatif des types et fonctions

Formule logique

```
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
```

Arbre de décision

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

Environnement

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

nœud de Binary Decision Diagram

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

Binary Decision Diagram

```
type bdd = (int * (bddNode list))
```

Arbre de décision numéroté

```
type decTreeNbr =
|DecLeafNbr of int*bool
|DecRootNbr of int*string*decTreeNbr*decTreeNbr
```

length

```
Argument : une liste l
Retourne : la longueur de l
```

isin

```
Arguments : une liste l et un élément q
Retourne : true si q est dans l, false sinon
```

unic

```
Argument : une liste l
Retourne : true si l ne contient aucun éléments en double, false sinon
```

getVars

```
Argument : une formule t
Retourne : la liste des variables de t
```

findvalue

```
Arguments : un environnement env et un string s
Retourne : le booléen du couple de env contenant s
```

evalFormula

```
Arguments : un environnement env et une fonction f
Retourne : la valeur de f dans env
```

buildDecTree

```
Argument : une fonction f
Retourne : l'arbre de décision de f
```

initBdd
Argument : un arbre de décision dec
Retourne : un Bdd et une liste d'arbres de décision contenant les feuilles contenues dans dec

isDecLeafIn
Arguments : deux arbres de décision dec1 et dec2
Retourne : true si dec2 est dans dec1, false sinon

grabDecTreeInt
Arguments : l'arbre dont on cherche l'adresse dec, un Bdd et une liste d'arbres de décision
Retourne : l'adresse du Bdd dont dec est le correspondant dans la liste d'arbres

buildBddFromDecTree
Arguments : un arbre de décision dec, le Bdd et la liste d'arbres de décision que l'on construit par induction
Retourne : le Bdd et la liste d'arbres de décision correspondant à dec

buildBdd
Argument : une fonction f
Retourne : le Bdd correspondant à f

splf
Arguments: une liste de nœuds, un nœud redundant et un nœud target
Retourne: la liste de nœuds dans laquelle les liens vers redundant ont été remplacés par des liens vers target

deleteRedundant
Arguments : deux listes de nœuds identiques
Retourne : la liste dont les nœuds inutiles ont été supprimés

simplifyBdd
Argument : un Bdd bdd
Retourne : le Bdd simplifié de bdd

addTrueFalse
Argument : une liste de listes l de booléens
Retourne : une liste contenant toutes les listes de l avec true devant et toutes les listes de l avec false devant

boolListsSizeN
Argument : un entier n positif
Retourne : la liste de toutes les listes de booléens de taille n

boolListsFormula
Argument : une formule f
Retourne : la liste de toutes les listes de booléens de taille le nombre de variables de f

boolListToEnv
Arguments : une formule f et une liste de booléens de taille le nombre de variables de f
Retourne : Un environnement de f

envLists
Argument : une formule f
Retourne : la liste de tous les environnements de f

isTautologyNoBdd
Argument : une formule f
Retourne : true si f est une tautologie, false sinon

isTautology
Argument : une formule f
Retourne : true si f est une tautologie, false sinon

areEquivalentNoBdd
Arguments : deux formules f1 et f2
Retourne : true si f1 $\equiv$ f2, false sinon

lengthBdd
Argument : un Bdd
Retourne : la longueur de sa liste de nœuds

replaceAdd
Arguments : une liste de nœuds et quatre entiers before, after, newG et newD
Retourne : la liste de nœuds dont les éléments ont été modifiés : si leur numéro était before alors on le change par after et on change ses successeurs par newG et newD. Sinon, si ses successeurs sont before on le supprime. Si un seul est before on le remplace par after

buildTwoDecTrees
Arguments : deux formules
Retourne : les arbres de décision des deux formules construits avec le même ordre de variables

bddlist
Argument : un Bdd
Retourne : sa liste de nœuds

areEquivalent
Arguments : deux formules f1 et f2
Retourne : true si f1 et f2 sont équivalentes, false sinon

dotBdd
Arguments : un nom de fichier nomfichier et un Bdd bdd
Retourne : écrit dans nomfichier la représentation graphique de bdd interprétable par dot

nbrTree
Argument : un arbre de décision dectree
Retourne : le nombre de feuilles et nœuds de dectree

addNbrTree
Argument : un arbre de décision dectree
Retourne : l'arbre de décision numéroté équivalent à dectree

dotDecTree
Arguments : un nom de fichier nomfichier et un arbre de décision dectree
Retourne : écrit dans nomfichier la représentation graphique de dectree interprétable par dot