UNIVERSITÉ PARIS 13 | USPC Université Sorbonne Paris Cité

Institut Galilée UNIVERSITÉ PARIS 13

Programmation fonctionnelle

TABLE DES MATIÈRES :

I - INTRODUCTION :

PRÉSENTATION DU LANGAGE OCAML :

Ocaml est un langage avec des traits fonctionnels et impératifs qui est enrichi par des constructions objets et un puissant système de modules. Il s'agit d'un langage fortement typé (avec inférence des types) ce qui en fait un langage sûr et facile à employer.

Toutefois, contrairement à la plupart des autres langages, il n'est pas nécessaire de préciser le type des variables que l'on utilise. En effet, Ocaml dispose d'un algorithme d'inférence de types qui lui permet de déterminer le type des variables à partir du contexte dans lequel elles sont employées.

L'inférence de types est un mécanisme qui permet à un compilateur ou un interpréteur de rechercher automatiquement les types associés à des expressions, sans qu'ils soient indiqués explicitement dans le code source
.

PRÉSENTATION DU PROJET :

Le but de ce devoir est d'implémenter en langage Ocaml un solveur booléen, c'est-à-dire un algorithme permettant de calculer l'ensemble des solutions d'un système d'équations booléennes.

Pour cela, on utilisera un type représentant les expressions booléennes nommé **eb.**

ce type sera défini en Ocaml de la manière suivante :

```
# type eb = V of int | TRUE | FALSE | AND of eb * eb | OR of eb * eb |
XOR of eb * eb | NOT of eb ;;
```

une expression de type eb peut contenir des variables V(i) (i est un nombre entier) , TRUE ou FALSE, elle peut contenir aussi d'autres expressions de type eb liés par l'un des connecteurs logiques « AND », « OR », « XOR » ou « NOT » .

Exemple : AND(V(1) , OR(V(2) , V(3)))

Afin de représenter une équation de type eb , on utilisera un type **equal** qu'on le définie comme suivant :

```
# type equal = EQUAL of eb * eb ;;
```

Exemple : EQUAL(OR (V(1) , V(2)) , TRUE)

L' algorithme de ce solveur booléen est composé principalement de trois étapes :

- Détermination de l'ensemble des variables du système d'équations , Cette étape consiste à récupérer les variables utilisées dans un système d'équations.
- Génération de tous les environnements possibles .Cette étape permet de déterminer tous les valeurs possibles prises par ces variables
- Évaluation de la satisfaction d'une équation donnée dans un environnement associé .

II - DÉFINITION DES FONCTIONS :

let rec union l1 l2 : eb list -> eb list -> eb list

C’est une fonction récursive qui renvoie l'union de deux listes prises en paramètre :

- L'union de deux listes vides, est une liste vide .
- Si l’une des deux listes est vide, on renvoie la liste non vide.
- Si le premier élément de la première liste est égal au premier élément de la deuxième liste, donc leur union sera l’union du reste de la première liste et le reste de la deuxième liste en incluant ce premier
élément dedans .
- Si le premier élément de la première liste n'égale pas au premier élément de la deuxième liste, donc le résultat sera l’union du reste de la première liste et le reste de la deuxième liste en incluant ces deux éléments dedans .

Exemple : union [V(1);V(2)] [V(3);V(4)] ;;

La fonction renvoie :

```
- : eb list = [V 1; V 3; V 2; V 4]
```

let rec variable equ : `eb -> eb list`

- Cette fonction prends en paramètre une expression booléenne, et renvoie les variables de cette équation :
- si l'expression vaut TRUE ou FALSE, la fonction renvoie une liste vide
- Si l'expression vaut une variable, la fonction renvoie une liste ne contenant que cette variable.
- Si l'expression s'ecrit sous forme de connecteurs logiques : « AND (a,b) , OR (a,b) , XOR (a,b) » la fonction renvoie **union (variable a) (variable b)** .
- Si l'expression s'écrit sous forme de NOT(a), la fonction renvoie **(variable a)** .

Exemple : `variable (AND(V(1),V(2))) ;;`

La fonction renvoie :

```
- : eb list = [V 1; V 2]
```

let variableEqu equ : `equal -> eb list`

- Cette fonction prend en paramètre une équation booléenne, et renvoie les variables du cote droit et celui du gauche de cette équation .

Exemple : `variableEqu (EQUAL(AND( V(1),V(2)),TRUE)) ;;`

La fonction renvoie :

```
- : eb list = [V 1; V 2]
```

let rec variableSys systemEqu : `equal -> eb list`

- Cette fonction prend en paramètre un système booléen, et renvoie les variables de tous les équations de ce système.

Exemple :
```
variableSys [EQUAL(AND( V(1),V(2)),TRUE);
  EQUAL(OR( V(1),V(2)),FALSE)] ;;
```

La fonction renvoie :

```
- : eb list = [V 1; V 2]
```

let rec environment list_var list_empty :

'a list -> ('a * eb) list -> ('a * eb) list list

- Cette fonction prend en paramètres :

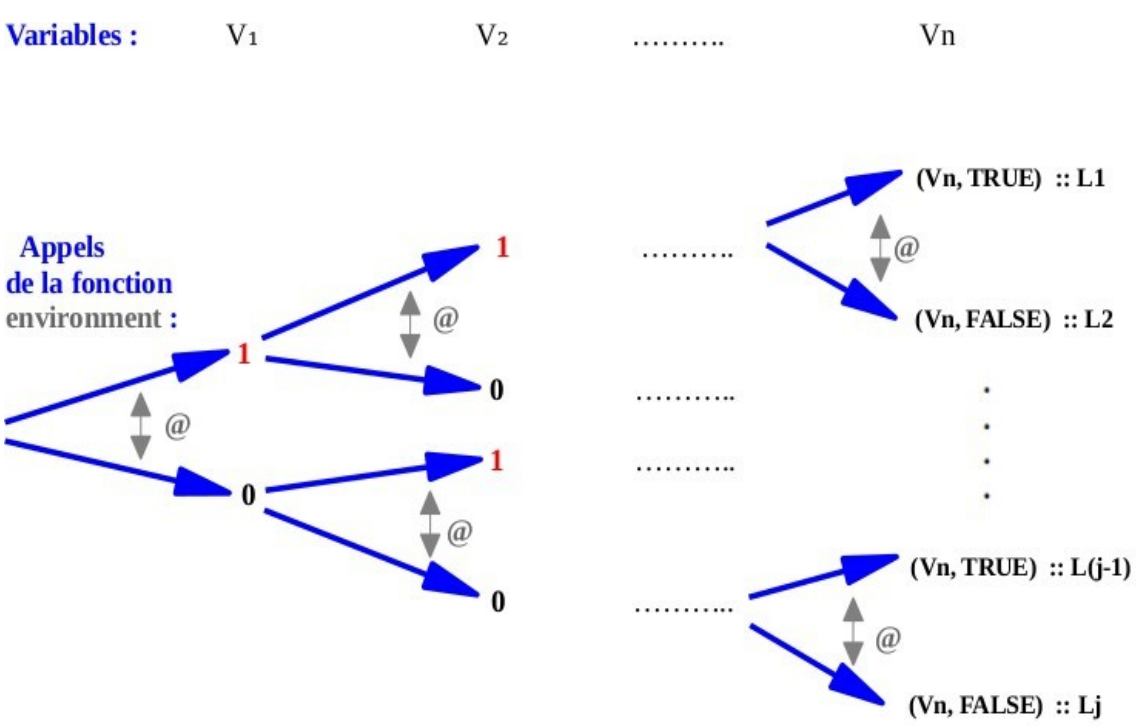
  list_var : La liste des variables de l'équation booléenne

  list_empty : Une liste vide dans laquelle on accumulera les couples (V(i),TRUE) et (V(i),FALSE).

- Si list_var est une liste vide, la fonction renvoie une liste vide .

- Si list_var contient un seul élément v elle renvoie [[(v,TRUE)] ; [(v,FALSE)]].

- si list_var contient plusieurs variables , le schéma suivant représente les appels successifs de la fonction **environment :**



1 représente l'appel environment l(i) ((Vi,TRUE) :: Lj)

0 représente l'appel environment l(i) ((Vi,FALSE) :: Lj)

- **l(i) = Vi :: ……. :: V(n-1) :: l(n)**
- **Lj est une liste qui accumule les couples (Vi, TRUE) ou (Vi, FALSE) d'un certain chemin d'appels de cette fonction.**

Avec $1 \leq i \leq n$ **(n est le nombre de variables) et** $1 \leq j \leq 2^i$

Exemple : environment [V(1);V(2)] [ ] ;;

La fonction renvoie :

```
- : (eb * eb) list list =
[[(V 2, TRUE); (V 1, TRUE)]; [(V 2, FALSE); (V 1, TRUE)];
 [(V 2, TRUE); (V 1, FALSE)]; [(V 2, FALSE); (V 1, FALSE)]]
```

let truth exp : `eb -> eb`

Cette fonction permet de définir les connecteurs logiques.

- Si l'expression donnée en paramètre vaut **TRUE**, le résultat sera **TRUE**.
- Si l'expression donnée en paramètre vaut **FALSE**, le résultat sera **FALSE**.
- Si l'expression donnée est sous forme d'expressions TRUE ou FALSE liés par l'un des connecteurs « **AND** » « **OR** » « **XOR** »
  « **NOT**» , alors :

AND		**Valeur retournée**
TRUE	**TRUE**	**TRUE**
TRUE	**FALSE**	**FALSE**
FALSE	**TRUE**	**FALSE**
FALSE	**FALSE**	**FALSE**

.

OR		**Valeur retournée**
TRUE	**TRUE**	**TRUE**
TRUE	**FALSE**	**TRUE**
FALSE	**TRUE**	**TRUE**
FALSE	**FALSE**	**FALSE**

XOR		Valeur retournée
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

NOT	Valeur retournée
TRUE	FALSE
FALSE	TRUE

Exemple : truth (AND(TRUE,FALSE)) ;;

La fonction renvoie :

```
- : eb = FALSE
```

let rec solve exp : `eb -> eb`

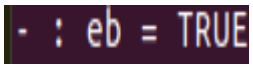
Cette fonction prend en paramètre une expression booléenne :

- Si l'expression vaut TRUE , la fonction renvoie TRUE
- Si l'expression vaut FALSE , la fonction renvoie FALSE
- Si l'expression s'ecrit sous forme de connecteurs logiques« AND (a,b) , OR (a,b) , XOR (a,b) , NOT(a) »

La fonction renvoie la valeur de la solution des expressions booléennes dedans le connecteur .

Exemple : `solve(AND(OR(TRUE,FALSE),TRUE)) ;;`

La fonction renvoie :



let solveEqu equ : equal -> bool

Cette fonction prend en paramètre une équation booléenne, et renvoie la valeur de vérité de cette équation en comparant la solution de ses deux membres.

Exemple :

```
solveEqu (EQUAL(AND(OR(TRUE,FALSE),TRUE),TRUE)) ;;
```

La fonction renvoie :

```
- : bool = true
```

let rec solveSys sys : equal list -> bool

Cette fonction prend en paramètres un système d'équations booléennes.

- Si le système est une liste vide , elle renvoie true.
- Sinon elle renvoie la valeur de vérité de la première équation en parallèle avec la solution du reste du système .

Exemple :

```
solveEqu
[EQUAL(AND(OR(TRUE,FALSE),TRUE),TRUE) ;EQUAL(OR(AN
D(TRUE,TRUE),FALSE),TRUE)] ;;
```

La fonction renvoie :

```
- : bool = true
```

let rec replace exp listValue : `eb -> (eb * eb) list -> eb`

Cette fonction prend en paramètres une expression booléenne et un environnement , elle remplace les variables de cette expression par cet environnement.

- Si la l'environnement est vide , elle renvoie l'expression.
- Si l'expression vaut TRUE (respectivement FALSE) , elle renvoie TRUE (respectivement FALSE) .
- Si l'expression est une variable et l'environnement est non vide, la fonction cherche la variable dans l'environnement et la remplace par la valeur associé dedans .
- Si l'expression est sous forme d'expressions booléennes liées par des connecteurs logiques , la fonction remplace l'environnement dans ces expressions booléennes.

Exemple :

```
replace (AND(OR(V(2),V(1)),TRUE)) [(V(1),TRUE);(V(2),FALSE)] ;;
```

La fonction renvoie :

```
- : eb = AND (OR (FALSE, TRUE), TRUE)
```

let rec replaceEqu equ listValue : eb -> (eb * eb) list -> eb

Cette fonction prend en paramètres une équation et un environnement, elle remplace cet environnent dans cette équation.

Exemple :

```
replaceEqu  (EQUAL(AND(OR(V(2),V(1)),TRUE),TRUE))
[(V(1),TRUE);(V(2),FALSE)]  ;;
```

La fonction renvoie :

```
- : equal = EQUAL (AND (OR (FALSE, TRUE), TRUE), TRUE)
```

let rec replaceSys systemEqu listValue :
equal list -> (eb * eb) list -> equal list

Cette fonction prend en paramètres un système d'équations booléennes et un environnement, elle remplace cet environnent dans ce système.

Exemple :

```
replaceSys
[EQUAL(AND(OR(V(2),V(1)),TRUE),TRUE);EQUAL(AND(OR(V(2),V
(1)),TRUE),TRUE)]  [(V(1),TRUE);(V(2),FALSE)]  ;;
```

La fonction renvoie :

```
- : equal list =
[EQUAL (AND (OR (FALSE, TRUE), TRUE), TRUE);
 EQUAL (AND (OR (FALSE, TRUE), TRUE), TRUE)]
```

let rec evaluation systemEqu envir :

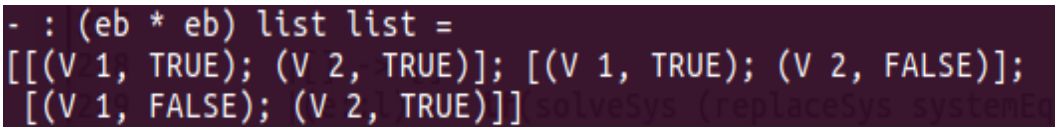
equal list -> (eb * eb) list list -> (eb * eb) list list

Cette fonction prend en paramètres un système d'équations booléennes et la liste de tous environnements possibles, elle renvoie la liste des environnement satisfaisant par ce système .

Exemple :

evaluation [EQUAL(AND(OR(V(2),V(1)),TRUE),TRUE);EQUAL(AND(OR(V(2),V(1)),TRUE),TRUE)] [[(V(1),TRUE);(V(2),TRUE)];[(V(1),TRUE);(V(2),FALSE)];[(V(1),FALSE);(V(2),TRUE)];[(V(1),FALSE);(V(2),FALSE)]] ;;

La fonction renvoie :



let projet systemEqu : equal list -> (eb * eb) list list

Cette fonction prend en paramètre un système d'équations booléennes et , elle renvoie les solutions de ce dernier .

Exemple :

```
projet  [EQUAL(OR(V(1),V(2)),TRUE)  ;
EQUAL(XOR(V(1),V(3)),V(2)) ;
EQUAL(NOT(AND(V(1),AND(V(2),V(3)))),TRUE  )]  ;;
```

La fonction renvoie :

```
[[(V 3, FALSE); (V 2, TRUE); (V 1, TRUE)];
 [(V 3, TRUE); (V 2, FALSE); (V 1, TRUE)];
 [(V 3, TRUE); (V 2, TRUE); (V 1, FALSE)]]
```

III – PROGRAMME :

ALGORITHME :

Union (l1,l2)

ENTRÉE: deux liste l1 et l2
SORTIE : l'union des deux listes l1 et l2

SI (l1 et l2) est vide ALORS
retourne liste Vide

SI l2 est vide ALORS
retourne l1
SI l1 est vide ALORS
retourne l2

SINON SI
(1er élément de l1 = 2eme élément de l2)
ALORS concatène (1er élément de l1 , union reste l1 reste l2)

SINON
concatène(1er élément l2,union(l1,reste l2))

FIN Fonction

Variable equ

ENTRÉE : une équation booléenne eb

SORTIE : la liste des variable de l'équation eb

DÉBUT

SINON SI eb = TRUE ALORS

retourne liste vide

SINON SI eb = FALSE ALORS

retourne liste vide

SI eb est une variable Vi ALORS

retourne liste de Vi

SINON SI eb = AND(p1,p2) ALORS

retourne union (variable p1 et variable p2)

SINON SI eb = OR(p1,p2) ALORS

rretourne union (variable p1 et variable p2)

SINON SI eb = XOR(p1,p2) ALORS

retourne union (variable p1 et variable p2)

SINON SI eb = NOT(p) ALORS

retourne variable p

FIN Fonction

variableEqu equ

ENTRÉE : une équation booléenne eb

SORTIE : la liste des variable de l'équation eb

DÉBUT

SI equ = EQUAL(a,b) ALORS

union (variable a) (variable b)

FIN SI

FIN FONCTION

variableSys systemEqu

ENTRÉE : systemEqu le système d'équation booléenne

SORTIE : Une liste des variables du système équation

DÉBUT

SI systemEqu= listeVide ALORS

renvoie listeVide

SI systemEqu= e :: resteListe ALORS

union (variableEqu e) (variableSys resteListe)

FIN SI

FIN FONCTION

environment list_var list_empty

ENTRÉE : list_var = la la liste des variable ; list_empty = un accumulateur sous forme de listeVide

SORTIE : La liste de tout les environnements possibles pour le système

note acc = accumulateur, (élément 1 :: reste) correspond au le premier élément et le reste de la liste

SI list_var = listeVide ALORS

RETOUR list_empty

SINON SI (élément 1 :: listeVide) ALORS ((élément 1 ,TRUE) :: list_empty :: listVide) concaténé ((élément 1 ,FALSE) :: list_empty :: listVide)

SINON SI (élément 1 :: resteListe) ALORS

environment resteListe ((élément 1 ,TRUE) :: list_empty)) concaténé environment resteList ((élément 1,FALSE) :: list_empty))

FIN SI

FIN FONCTION

solve exp

ENTRÉE : une expression booléenne exp (sans variable)

SORTIE : la valeur de l'expression exp

DÉBUT

SI exp est une variable Vi ALORS

retourne FALSE

SINON SI exp = TRUE ALORS

retourne TRUE

SINON SI exp = FALSE ALORS

retourne FALSE

SINON SI exp = AND(a,b) ALORS

retourne truth (AND (solve a , solve b

)) SINON SI exp = OR(a,b) ALORS

retourne truth (OR (solve a , solve b))

SINON SI exp = XOR(a,b) ALORS

retourne truth (XOR (solve a , solve b))

SINON SI exp = NOT(p) ALORS truth (NOT(solve p))

FIN SI

FIN FONCTION

solveEqu equ

ENTRÉE : une équation booléenne equ

SORTIE : la valeur de l'expression equ

DÉBUT

SI equ = EQUAL(a,b) ALORS

solve a = solve

b FIN SI

FIN FONCTION

solveSys sys

ENTRÉE : une expression booléenne sys

SORTIE : la valeur de l'expression sys

DÉBUT

SI sys est une listeVide

ALORS retourne true

SINON SI sys = EQUAL(a,b) :: resteListe ALORS

(solveEqu (EQUAL(a,b)) ET (solveSys resteListe

) FIN SI

FIN FONCTION

replace exp listValue

ENTRÉE : exp : une expression booléenne ; listValue : un environnement

SORTIE : une expression booléenne

DÉBUT

SI exp = a ET listValue est une listeVide ALORS

retourne a

SINON SI exp = TRUE ET listValue

ALORS retourne TRUE

SINON SI exp = FALSE ET listValue ALORS
retourne FALSE

SINON SI (exp = Vi) ET listValue = (x,y)::resteListe ALORS
SI (Vi = x)

ALORS y

SINON replace Vi resteListe

FIN SI

SINON SI (exp = AND(a,b)) ET listValue ALORS

retourne AND (replace a listValue , replace b listValue)

SINON SI (exp = OR(a,b)) ET listValue ALORS
retourne OR (replace a listValue , replace b listValue)

SINON SI (exp =XOR(a,b)) ET listValue ALORS
retourne XOR (replace a listValue , replace b listValue)

SINON SI (exp=NOT(p)) ET listValue ALORS

retourne NOT (replace p listValue)

FIN SI

FIN FONCTION

replaceEqu equ listValue

ENTRÉE : exp : une expression booléenne ; listValue : un environnement

SORTIE : une équation booléenne

DÉBUT

SI (equ = EQUAL(a,b)) ET listValue ALORS

retourne EQUAL((replace a listValue) , (replace b listValue))

FIN SI

FIN FONCTION

replaceSys systemEqu listValue

ENTRÉE : systemEqu : un systeme d'équation ; listValue : un environnement

SORTIE : un système d'équations

DÉBUT

SI ((systemEqu est une listeVide) ET (quel que soit l'environnement)) ALORS

retourne listeVide

SINON SI systemEqu = EQUAL(a,b) :: resteListe ET listValue ALORS

retourne (replaceEqu (EQUAL(a,b)) listValue) :: (replaceSys

resteListe listValue)

FIN SI

FIN FONCTION

evaluation systemEqu envir

ENTRÉE : systemEqu : un systeme d'équation ; envir : un environnement

SORTIE : les environnements satisfaisants les système d'équation booléenne .

DÉBUT

SI l'environnement est une listeVide ALORS

retourne listeVide

SINON SI (1er élément :: resteListe) ALORS

SI (solveSys (replaceSys systemEqu 1er élément)) ALORS retourne [1er élément] concaténé (evaluation systemEqu resteListe)

SINON SI (evaluation systemEqu resteListe)

FIN SI

FIN SI

FIN FONCTION

CODE :

```
(* Définition du type de l'expression booléene *)

type eb = V of int | TRUE | FALSE | AND of eb * eb | OR of eb * eb | XOR
of eb * eb | NOT of eb ;;

(* Définition du type equal permettant de présenter une équation booléene
*)

type equal = EQUAL of eb * eb ;;



(* la fonction "union" prend en paramètre deux listes de variables
entières ,et renvoie l'union de ces deux listes *)


let rec union l1 l2 =

        match (l1,l2)

        with

        |([],[]) -> []
        |(l1,[])-> l1
        |([],l2)-> l2

        |(V(i)::ll1 , V(j)::ll2) ->
        if i=j then V(i)::(union ll1
        ll2) else V(i)::V(j)::(union ll1
        ll2)

        |e::l,f::k -> [] ;;
```

```
(* la fonction "variable" prend en paramètre une expression booléene , et récupère les variables de cette expression *)


let rec variable equ =

        match equ with

        |TRUE->[]
        |FALSE->[]
        |V(a) -> [V(a)]
        |AND(p1,p2)-> union (variable p1) (variable p2)
        |OR(p1,p2)-> union (variable p1) (variable p2)
        |XOR(p1,p2)-> union (variable p1) (variable p2)
        |NOT(p)-> variable p  ;;



(* la fonction "variableEqu" prend en paramètre une équation booléene , et récupère les variables du côté droit et celui du gauche de cette équation *)


let variableEqu equ =

        match equ with

        |EQUAL(a,b) -> union (variable a) (variable b) ;;
```

(* la fonction "variableSys" prend en paramètre un système d'équations booléenes , et récupère tous les variables de ce système *)

```
let rec variableSys systemEqu

        = match systemEqu with

        |[]->[]
        |e::l -> union (variableEqu e) (variableSys l) ;;
```

(* la fonction "environment" prend en paramètre une liste de variables entières et une liste vide qui nous sera utile pour accumuler les couples (V(i),TRUE) et (V(i),FALSE) , elle renvoie tous les environements possibles *)

```
let rec environment list_var list_empty =

        match list_var with

        |[]->[]
        |v::[]-> ( ( (v,TRUE)::list_empty)::[]) @ ( ( (v,FALSE)::list_empty)::[] )
        |v::l ->  (environment l ((v,TRUE)::list_empty)) @ (environment l
((v,FALSE)::list_empty) )  ;;
```

(* la fonction "truth" prend en paramètre une expression booléen ne contenant qu'un connecteur logique et des TRUE ou FALSE , elle renvoie la valeur de cette expression en se basant sur les cas initiales *)

```
let truth exp =

        match exp with

        |TRUE -> TRUE
        |FALSE -> FALSE
        |AND(TRUE,TRUE) -> TRUE |AND(TRUE,FALSE) -> FALSE  |
AND(FALSE,TRUE) -> FALSE |AND(FALSE,FALSE) -> FALSE
        |OR(TRUE,TRUE) -> TRUE |OR(TRUE,FALSE) -> TRUE  |OR(FALSE,TRUE)
-> TRUE  |OR(FALSE,FALSE) -> FALSE
        |XOR(TRUE,TRUE) -> FALSE |XOR(TRUE,FALSE) -> TRUE  |
XOR(FALSE,TRUE) -> TRUE |XOR(FALSE,FALSE) -> FALSE
        |NOT(TRUE) -> FALSE |NOT(FALSE) -> TRUE ;;
```

```
(* la fonction "solve" prend en paramètre une expression booléenne ne contenant pas de variables , elle renvoie la valeur de cette expression en utilisant la fonction "truth" *)

let rec solve exp =

        match exp with

        |V(i) -> FALSE
        |TRUE -> TRUE
        |FALSE -> FALSE
        |AND(a,b) -> truth (AND(solve a, solve b))
        |OR(a,b) -> truth (OR(solve a, solve b))
        |XOR(a,b) -> truth (XOR(solve a, solve b))
        |NOT(p) -> truth (NOT(solve p)) ;;




(* la fonction "solveEqu" prend en paramètre une équation booléenne ne contenant pas de variables , elle renvoie la valeur de vérité de l'équation en utilisant la fonction "solve" *)


let solveEqu equ =

        match equ with

        |EQUAL(a,b) -> solve a = solve b ;;
```

```
(* la fonction "solveSys" prend en paramètre un système d'équations booléennes ne contenant pas de variables , elle renvoie la valeur de vérité de ce système en utilisant la fonction "solveEqu" *)

let rec solveSys sys =

        match sys with

        |[]->true
        |EQUAL(a,b)::l -> (solveEqu (EQUAL(a,b))) && (solveSys l);;




(* la fonction "replace" prend en paramètre une expression booléenne et un environement , elle remplace cet environement dans l'expression *)

let rec replace exp listValue =

        match (exp,listValue)

        with

        |(a,[]) -> a
        |(TRUE,listValue) -> TRUE
        |(FALSE,listValue) -> FALSE

        |(V(i),(x,y)::l) -> if (V(i) = x)
                            then y
                            else replace (V(i)) l

        |(AND(a,b),listValue) -> AND(replace a listValue, replace b listValue)
        |(OR(a,b),listValue) -> OR(replace a listValue, replace b listValue)
        |(XOR(a,b),listValue) -> XOR(replace a listValue, replace b listValue)
        |(NOT(p),listValue) -> NOT(replace p listValue)  ;;
```

```
(* la fonction "replaceEqu" prend en paramètre une équation booléenne et un environement , elle remplace cet environement dans l'équation *)


let replaceEqu equ listValue =

        match equ with

         |(EQUAL(a,b)) -> EQUAL((replace a listValue) , (replace b listValue))
                                                                    ;;




(* la fonction "replaceSys" prend en paramètre un système d'équations booléennes et un environement , elle remplace cet environement dans ce système *)


let rec replaceSys systemEqu listValue =

        match systemEqu with

        |[]-> []
        |(EQUAL(a,b)::l) -> (replaceEqu (EQUAL(a,b)) listValue):: (replaceSys l listValue) ;;
```

(* la fonction "evaluation" prend en paramètre un système d'équations booléennes et une liste contenant tous les environements possibles , elle renvoie une liste contenante tous les environements satisfaisants ce système *)

```
let rec evaluation systemEqu envir =

        match envir with

        |[] -> []
        |(e::l) -> if(solveSys (replaceSys systemEqu e)) then [e]@(evaluation systemEqu l)
                                else (evaluation systemEqu l) ;;
```

(* la fonction "projet" prend en paramètre un système d'équations booléennes , elle renvoie tous les solutions ce système*)

```
let projet systemEqu = evaluation systemEqu (environment (variableSys systemEqu) []) ;;
```

JEU D'ESSAI :

projet [EQUAL(OR(V(1),V(2)),TRUE) ;
EQUAL(XOR(V(1),V(3)),V(2)) ;
EQUAL(NOT(AND(V(1),AND(V(2),V(3)))),TRUE)] ;;

```
# #use "Projet_Ocaml.ml" ;;
type eb =
    V of int
  | TRUE
  | FALSE
  | AND of eb * eb
  | OR of eb * eb
  | XOR of eb * eb
  | NOT of eb
type equal = EQUAL of eb * eb
val union : eb list -> eb list -> eb list = <fun>
val variable : eb -> eb list = <fun>
val variableEqu : equal -> eb list = <fun>
val variableSys : equal list -> eb list = <fun>
val environment : 'a list -> ('a * eb) list -> ('a * eb) list list = <fun>
File "Projet_Ocaml.ml", line 118, characters 1-402:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
V _
val truth : eb -> eb = <fun>
val solve : eb -> eb = <fun>
val solveEqu : equal -> bool = <fun>
val solveSys : equal list -> bool = <fun>
val replace : eb -> (eb * eb) list -> eb = <fun>
val replaceEqu : equal -> (eb * eb) list -> equal = <fun>
val replaceSys : equal list -> (eb * eb) list -> equal list = <fun>
val evaluation : equal list -> (eb * eb) list list -> (eb * eb) list list =
  <fun>
val projet : equal list -> (eb * eb) list list = <fun>
- : (eb * eb) list list =
[[(V 3, FALSE); (V 2, TRUE); (V 1, TRUE)];
 [(V 3, TRUE); (V 2, FALSE); (V 1, TRUE)];
 [(V 3, TRUE); (V 2, TRUE); (V 1, FALSE)]]
#
```

```
projet  [EQUAL(AND(V(1),V(2)),TRUE)  ;
EQUAL(XOR(V(1),V(3)),V(2))  ;
EQUAL(OR(V(1),OR(V(2),V(3))),TRUE  )]  ;;
```

```
# #use "Projet_Ocaml.ml" ;;
type eb =
    V of int
  | TRUE
  | FALSE
  | AND of eb * eb
  | OR of eb * eb
  | XOR of eb * eb
  | NOT of eb
type equal = EQUAL of eb * eb
val union : eb list -> eb list -> eb list = <fun>
val variable : eb -> eb list = <fun>
val variableEqu : equal -> eb list = <fun>
val variableSys : equal list -> eb list = <fun>
val environment : 'a list -> ('a * eb) list -> ('a * eb) list list = <fun>
File "Projet_Ocaml.ml", line 118, characters 1-402:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
V _
val truth : eb -> eb = <fun>
val solve : eb -> eb = <fun>
val solveEqu : equal -> bool = <fun>
val solveSys : equal list -> bool = <fun>
val replace : eb -> (eb * eb) list -> eb = <fun>
val replaceEqu : equal -> (eb * eb) list -> equal = <fun>
val replaceSys : equal list -> (eb * eb) list -> equal list = <fun>
val evaluation : equal list -> (eb * eb) list list -> (eb * eb) list list =
  <fun>
val projet : equal list -> (eb * eb) list list = <fun>
- : (eb * eb) list list = [[(V 3, FALSE); (V 2, TRUE); (V 1, TRUE)]]
#
```

```
projet  [EQUAL(AND(V(1),V(2)),FALSE)  ;
EQUAL(OR(V(1),V(3)),V(2))  ;
EQUAL(OR(V(1),AND(V(2),V(3))),TRUE  )]  ;;
```

```
# #use "Projet_Ocaml.ml" ;;
type eb =
    V of int
  | TRUE
  | FALSE
  | AND of eb * eb
  | OR of eb * eb
  | XOR of eb * eb
  | NOT of eb
type equal = EQUAL of eb * eb
val union : eb list -> eb list -> eb list = <fun>
val variable : eb -> eb list = <fun>
val variableEqu : equal -> eb list = <fun>
val variableSys : equal list -> eb list = <fun>
val environment : 'a list -> ('a * eb) list -> ('a * eb) list list = <fun>
File "Projet_Ocaml.ml", line 118, characters 1-402:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
V _
val truth : eb -> eb = <fun>
val solve : eb -> eb = <fun>
val solveEqu : equal -> bool = <fun>
val solveSys : equal list -> bool = <fun>
val replace : eb -> (eb * eb) list -> eb = <fun>
val replaceEqu : equal -> (eb * eb) list -> equal = <fun>
val replaceSys : equal list -> (eb * eb) list -> equal list = <fun>
val evaluation : equal list -> (eb * eb) list list -> (eb * eb) list list =
  <fun>
val projet : equal list -> (eb * eb) list list = <fun>
- : (eb * eb) list list = [[(V 3, TRUE); (V 2, TRUE); (V 1, FALSE)]]
#
```