# TP de Logique des Propositions (HLIN 402)

Licence Informatique - Université de Montpellier
M. Leclère - leclere@lirmm.fr

**Résumé**

L'objectif de ce TP est d'implanter un raisonneur pour la logique des propositions en Racket (une famille de langages dérivés de Lisp). On utilisera en particulier son environnement de développement DrRacket. Toutes les focntions de ce TP ont été faites en choisissant dans le menu "Language" : *The Racket Language*. On vous fournit un format de représentation des formules bien formées de la logique des propostions et un ensemble de fonctions permettant d'avoir accès aux différents éléments syntaxique de ces formules. On vous demande dans un premier temps de doter ces formules de leur sémantique booléenne et de proposer des fonctions pour résoudre les différents problèmes de la logique des propositions : satisfiabilité, équivalence, conséquence logique... Dans un second temps, vous serez amené à implanter la méthode de résolution.

Pour réaliser ce TP vous ne disposez que de 8 séances aussi il faut absolument que vous travailliez en dehors des TP pour respecter l'échéancier suivant :

- La séance 1 doit être consacrée aux questions Q1 à Q11 (la question Q6 étant optionnelle).
- La séance 2 doit être consacrée aux questions Q12 à Q16 .
- La séance 3 doit être consacrée aux questions Q17 à Q24.
- La séance 4 doit être consacrée aux questions Q25 à Q30.
- La séance 5 doit être consacrée aux questions Q31 à Q38
- La séance 6 doit être consacrée aux questions Q39 à Q40.
- La séance 7 doit être consacrée à la question Q41 et à l'évaluation de TP.

## 1 Représentation et manipulation des propositions

Pour représenter les différents éléments du langage de la logique des propositions on utilise les valeurs du type `symbol` Scheme (rien à voir avec les symboles propositionnels) avec les conventions suivantes :

- les constantes logiques : `Top` pour $\top$ et `Bot` pour $\bot$ ;
- les connecteurs : `!` pour $\neg$, `^` pour $\wedge$, `v` pour $\vee$, `->` pour $\rightarrow$, `<->` pour $\leftrightarrow$,
- les symboles propositionnels : n'importe quelle autre valeur du type `symbol` (exemple : `a`, `b`, `p`, `toto`, `Jean`, `GAGNE`).

Une formule bien formée sera donc soit réduite à une valeur de `symbol` représentant une constante logique ou un symbole propositionnel, soit une liste de deux ou trois éléments (selon l'arité du connecteur) dont le premier élément est le connecteur racine et les autres éléments ses opérandes. Ainsi on déclare $F = p$ par `(define F 'p)`, $G = \neg toto$ par `(define G '(! toto))` et $H = a \wedge c \leftrightarrow \neg b \vee (c \rightarrow \bot \wedge \top)$ par `(define H '(<-> (^ a c) (v (! b) (-> c (^ Bot Top)))))`.

On dispose de plus des fonctions suivantes de manipulation des formules bien formées :

- `fbf?` : $ANY \rightarrow Bool$ qui teste si son paramètre est une formule bien formée ;
- `symbProp?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée est réduite à un symbole propositionnel ;
- `top?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée est réduite à la contante logique $\top$ ;
- `bot?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée est réduite à la contante logique $\bot$ ;
- `atomicFbf?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée est atomique (cas de base du langage) ;
- `negFbf?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée a pour connecteur racine une négation ;
- `conBinFbf?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée a pour connecteur racine un connecteur binaire ;
- `conjFbf?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée a pour connecteur racine une conjonction ;
- `disjFbf?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée a pour connecteur racine une disjonction ;
- `implFbf?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée a pour connecteur racine une implication ;
- `equiFbf?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée a pour connecteur racine une équivalence ;
- `conRac` : $PROP(S) \rightarrow Symbol$ qui retourne le connecteur racine d'une formule bien formée non réduite à un atome ;
- `fils` : $PROP(S) \rightarrow PROP(S)$ qui étant donnée une formule bien formée dont le connecteur racine est une négation retourne sa sous-formule fille ;
- `filsG` : $PROP(S) \rightarrow PROP(S)$ qui étant donnée une formule bien formée dont le connecteur racine est binaire retourne sa sous-formule fille gauche ;
- `filsD` : $PROP(S) \rightarrow PROP(S)$ qui étant donnée une formule bien formée dont le connecteur racine est binaire retourne sa sous-formule fille droite ;

**Q 1** Définir les formules suivantes dans la représentation Scheme proposée (on rappelle qu'il faut utiliser un *quote* pour bloquer l'évaluation d'une expression) :

$$F1 = a \wedge b \leftrightarrow \neg a \vee b$$

$$F2 = \neg(a \wedge \neg b) \vee \neg(a \rightarrow b)$$

$$F3 = \neg(a \rightarrow a \vee b) \wedge \neg\neg(a \wedge (b \vee \neg c))$$

$$F4 = (\neg a \vee b \vee d) \wedge (\neg d \vee c) \wedge (c \vee a) \wedge (\neg c \vee b) \wedge (\neg c \vee \neg b) \wedge (\neg b \vee d)$$

**Q 2** Vérifier à l'aide de la fonction `fbf?` que vos définitions de formules sont correctes.

# 2 Syntaxe des propositions

**Q 3** Écrire la fonction `nbc` qui retourne le nombre de connecteurs d'une fbf (nombre d'occurrences), en complétant la fonction suivante :

```
(define (nbc F)
   (cond ((atomicFbf? F)  ....)
         ((negFbf? F)     ....)
         (else             ....)))
```

**Q 4** Écrire la fonction `prof` qui retourne la profondeur d'une fbf.

**Q 5** Écrire la fonction `ensSP` qui retourne l'ensemble des symboles propositionnels d'une proposition donnée. On représentera les ensembles par de simples listes sans doublon. Par exemple, l'ensemble $\{p, q, r\}$ est représenté par la liste `( p q r )` et l'ensemble vide est simplement représenté par la liste vide `()`. Vous utiliserez les fonctions Scheme de manipulation des ensembles (ordonnés) :

- `(set-member s e)` qui retourne vrai si `e` est un élément de l'ensemble `s` ;
- `(set-empty? s)` qui retourne vrai si l'ensemble `s` est vide ;
- `(subset? s1 s2)` qui retourne vrai si `s1` est un sous-ensemble de `s2` ;
- `(set=? s1 s2)` qui retourne vrai si les ensembles `s1` et `s2` sont égaux (indépendamment de l'ordre des éléments) ;
- `(set-add s e)` qui retourne un ensemble contenant les éléments de `s` et l'élément `e` ;
- `(set-remove s e)` qui retourne un ensemble contenant les éléments de `s` sans l'élément `e` ;
- `(set-first s)` qui retourne le premier élément de l'ensemble `s` ;
- `(set-rest s)` qui retourne un ensemble constitué de tous les éléments de `s` sauf le premier ;
- `(set-union s1 s2)` qui retourne un ensemble constitué de l'union des ensembles `s1` et `s2` ;
- `(set-intersect s1 s2)` qui retourne un ensemble constitué de l'intersection des ensembles `s1` et `s2` ;
- `(set-subtract s1 s2)` qui retourne un ensemble constitué des éléments de `s1` qui n'appartiennent pas à `s2` ;
- `(set-count s)` qui retourne le nombre d'éléments de l'ensemble `s`.

**Q 6** *Optionnel* Écrire une fonction `affiche` qui prend en donnée une fbf et affiche cette formule sous forme infixée complètement parenthésée (écriture classique). Par exemple `(affiche F1)` retournera la chaîne $((a \wedge b) \leftrightarrow \neg(a \vee b))$. On utilisera la fonction `display` pour les affichages.

# 3 Sémantique des propositions

On représentera une interprétation par une liste de paires pointées `(symbole . valeur)`. Par exemple, l'inteprétation $I$ telle que $I(a) = 0$ et $I(b) = 1$ peut être définie par `(define I '( (a . 0) (b . 1)))`. On rappelle que les fonctions `(car pp)`, `(cdr pp)` et `(cons e1 e2)` permettent respectivement d'obtenir le premier élément d'une paire pointée, le deuxième élément d'une paire pointée, et de fabriquer une paire pointée à partir de ses deux éléments.

**Q 7** Définir les 3 interprétations $I1$, $I2$, $I3$ suivantes : $I1(a) = I1(c) = 1$ et $I1(b) = 0$, $I2(a) = I2(b) = I2(c) = 0$, $I3(a) = I3(b) = I3(c) = 1$.

**Q 8** Écrire la fonction `intSymb` qui retourne la valeur d'interprétation (`0` ou `1`) d'un symbole propositionnel donné dans une interprétation donnée (on supposera que ce symbole propositionnel apparaît dans la structure d'interprétation). Exemple : `(intSymb 'b I1)` doit retourner 0. Compléter la fonction :

```
(define (intSymb s I)
   (cond ((eq? (car (set-first I)) s) (cdr (set-first I)))
         (else         ....)))
```

**Q 9** Écrire les fonctions d'interprétation (unaires, binaires ou 0-aire) des connecteurs et constantes logiques : `intNeg`, `intAnd`, `intOr`, `intImp`, `intEqu`, `intTop`, `intBot`. Par exemple, `intAnd` peut s'écrire : `(define (intAnd v1 v2) (* v1 v2))`

**Q 10** Écrire la fonction `valV` qui calcule la valeur de vérité d'une formule $F$ pour une interprétation $I$ **supposée complète** pour $F$ (c'est-à-dire dont tous les symboles de $F$ ont une interprétation dans $I$).

**Q 11** Écrire un prédicat `modele?` qui, étant donné une inteprétation et une fbf, retourne vrai si l'interprétation est un modèle de la formule.

# 4 Satisfiabilité et validité d'une proposition

Afin d'étudier les propriétés sémantiques des propositions, on a besoin de considérer l'ensemble de toutes les interprétations des symboles propositionnels d'une formule donnée. Un ensemble d'inteprétations est donc une liste de listes de paires pointées.

**Q 12** Définir l'ensemble `EI` de toutes les interprétations des symboles propositionnels $p$ et $q$ (c'est une liste de 4 listes de 2 paires pointées).

Dans la suite, vous serez amenez à utiliser les fonctions `map`, `andmap`, `ormap`, `filter` et les fonctions anonymes (les *lambda expression*) qui permettent d'appliquer une fonction unaire $f$ (qui doit retourner un booléen pour `andmap`, `ormap` et `filter`) à chaque élément d'une liste $(e_1 e_2 \dots e_n)$ pour calculer resp. $(f(e_1)f(e_2)\dots f(e_n))$, $f(e_1) \wedge f(e_2) \wedge \dots \wedge f(e_n)$, $f(e_1) \vee f(e_2) \vee \dots \vee f(e_n)$, et la sous-liste de $(e_1 e_2 \dots e_n)$ telle que $f(e_i)$ est vraie. Par exemple :

- `(map (lambda (e) (* e 2)) '(1 2 5 6 9))` retourne `(2 4 10 12 18)` ;
- `(map (lambda (e) (= (remainder e 2) 0)) '(1 2 5 6 9))` retourne `(#f #t #f #t #f)` (vrai ssi il est pair) ;
- `(andmap (lambda (e) (= (remainder e 2) 0)) '(1 2 5 6 9))` retourne `#f` (tous ne sont pas pairs) ;
- `(ormap (lambda (e) (= (remainder e 2) 0)) '(1 2 5 6 9))` retourne `#t` (il y a au moins un pair dans la liste) ;
- `(filter (lambda (e) (= (remainder e 2) 0)) '(1 2 5 6 9))` retourne `(2 6)` (ce sont les entiers pairs de la liste) ;

**Q 13** Écrire une fonction `ensInt` qui prend en donnée un ensemble de symboles propositionnels et retourne l'ensemble de toutes les interprétations de ces symboles propositionnels. Vous utiliserez le `let` qui permet de nommer une expression (et évite ainsi de l'évaluer plusieurs fois) et le `append` qui permet de concaténer deux listes.

*Indication : lorsqu'il n'y a pas de symbole propositionnel, il n'y a qu'une seule interprétation possible, l'inteprétation vide, donc on retourne un ensemble contenant uniquement l'interprétation vide :* `(())`*. S'il y en a au moins un, il est judicieux de calculer récursivement l'ensemble* `EI` *des interprétations de tous les symboles sauf le premier, puis de prendre en compte le premier symbole en ajoutant à chaque interprétation de* `EI` *l'interprétation du premier symbole (une fois à* 0 *et une fois à* 1*). Compléter la fonction :*

```
(define (ensInt ensSymb)
   (if (set-empty? ensSymb)  '(())
                             (let ( (EI  ...) )
                                  (append (map (lambda (I) (set-add I ...)) EI)
                                          (map (lambda (I) (set-add I ...)) EI)))))
```

**Q 14** Écrire un prédicat `satisfiable?` qui retourne *true* si et seulement si une fbf donnée est satisfiable. Vous utiliserez la fonction `ormap`. Testez votre prédicat sur les propositions $a, \neg a, (a \wedge b), ((a \wedge b) \wedge \neg a), F_1, F_2, F_3, F_4$.

**Q 15** Écrire un prédicat `valide?` qui retourne *true* si et seulement si une fbf donnée est valide. Vous utiliserez la fonction `andmap`. Testez votre prédicat sur les propositions $a, \neg a, (a \vee b), ((a \vee b) \vee \neg a), F_1, F_2, F_3, F_4$.

**Q 16** Écrire un prédicat `insatisfiable?` qui retourne *true* si et seulement si une fbf donnée est insatisfiable. Vous utiliserez la fonction `filter`.

# 5 Equivalence et conséquence logique

**Q 17** Écrire **deux** versions d'un prédicat `equivalent?` qui teste si deux fbf données sont sémantiquement équivalentes : l'une `equivalent1?` s'appuiera sur la définition du cours (elles ont les mêmes valeurs de vérité pour toutes les interprétations), l'autre `equivalent2?` exploitera le lien entre équivalence sémantique et validité. Faire des tests ! En particulier $((a \vee b) \vee \neg a) \equiv \neg((c \wedge d) \wedge \neg c)$, $(a \vee \neg a) \equiv \neg(c \wedge (b \wedge \neg b))$ et $a \not\equiv b$.

**Q 18** Écrire un prédicat `consequence2?` qui étant donnée deux propositions $F1$ et $F2$ retourne *true* si $F2$ est conséquence logique de $F1$. Vérifier que $a \models (a \vee b)$, $a \not\models (a \wedge b)$, $((a \vee b) \vee \neg a) \models \neg((c \wedge d) \wedge \neg c)$, $((a \wedge b) \wedge \neg a) \models (c \vee d)$.

Pour la définition générale de la conséquence logique, on considère des ensembles de fbf naturellement représentés par une liste de fbf.

**Q 19** Écrire la fonction `ensSPallFbf` qui retourne l'ensemble des symboles propositionnels d'un ensemble de propositions donné, extension de la fonction `ensSP` à des ensembles de formules.

**Q 20** Écrire un prédicat `modeleCommun?` qui retourne *true* si une interprétation donnée est modèle d'un ensemble de propositions donné, extension du prédicat `modele?` à des ensembles de formules.

**Q 21** Écrire un prédicat `contradictoire?` qui retourne *true* si et seulement si un ensemble de propositions est contradictoire.

**Q 22** Écrire un prédicat `consequence?` prenant en donnée un ensemble de formules $\{f_1, f_2, ...f_n\}$ et une fbf $f$ et retournant true si $f$ est conséquence logique de $f_1...f_n$ (c'est à dire $\{f_1, f_2, ...f_n\} \models f$). Vérifier que $\{a \wedge b, \neg a, b \rightarrow d\} \models c \rightarrow d$.

**Q 23** Proposer une deuxième version de ce prédicat `consequenceV?` exploitant le lien entre conséquence logique et validité. Vous aurez certainement besoin d'écrire une fonction intermédiaire `conjonction` qui étant donné un ensemble de formules retourne la formule composée de la conjonction de toutes ces formules.

**Q 24** Proposer une troisième version de ce prédicat `consequenceI?` exploitant le lien entre conséquence logique et insatisfiabilité.

# 6 Mise sous forme conjonctive

Les 5 premières questions visent à fournir les transformations de fbf permettant un passage à la forme conjonctive. Pour ces fonctions, il faut raisonner sur l'arbre syntaxique associé à la formule. La 6$^e$ vise à fournir cette fonction. Attention, la 5$^e$ fonction (`distOu`) est particulièrement délicate. Ex. : `(distOu '(^ (^ a (! b)) (v c (v (! d) (^ e f)))) )` doit retourner `(^ (^ a (! b)) (^ (v c (v (! d) e)) (v c (v (! d) f))))`.

**Q 25** Écrire une fonction récursive `oteEqu` qui prend en paramètre une fbf et retourne une fbf logiquement équivalente qui ne contient pas de connecteur $\leftrightarrow$. Rappel : $(A \leftrightarrow B) \equiv ((A \rightarrow B) \wedge (B \rightarrow A))$

**Q 26** Écrire une fonction récursive `oteImp` qui prend en paramètre une fbf et retourne une fbf logiquement équivalente qui ne contient pas de connecteur $\rightarrow$. Rappel : $(A \rightarrow B) \equiv (\neg A \vee B)$

**Q 27** Écrire une fonction récursive `oteCste` qui prend en paramètre une fbf et retourne une fbf logiquement équivalente qui ne contient pas de constante logique. Rappel : $\top \equiv (\neg p \vee p)$ et $\bot \equiv (\neg p \wedge p)$

**Q 28** Écrire une fonction récursive `redNeg` qui prend en paramètre une fbf ne contenant pas de connecteur $\leftrightarrow$ et $\rightarrow$ et retourne une fbf logiquement équivalente dont la négation ne porte que sur les symboles propositionnels. Rappel : $\neg\neg A \equiv A$, $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$, $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$. Compléter la fonction :

```
(define (redNeg F)
(cond ((symbProp? F)     ...) ; cas d'un symbole propositionnel
      ((not (negFbf? F)) ...) ; cas d'un connecteur racine ^ ou v
      (else  ; cas de la négation en connecteur racine ==> on regarde son fils
          ((symbProp? (fils F))   ...) ; littéral négatif
          ((negFbf? (fils F))  ...) ; deux négations ==> équivalence de la double négation
          ((conjFbf? (fils F))  ...) ; négation d'une conjonction ==> équivalence de De Morgan
          (else  ...))))) ; négation d'une disjonction ==> équivalence de De Morgan
```

**Q 29** Écrire une fonction récursive `distOu` qui prend en paramètre une fbf composée de littéraux connectés par des $\wedge$ et $\vee$ et retourne une fbf logiquement équivalente sous forme conjonctive (i.e. conjonction de disjonctions de littéraux). Rappel : $(A \vee (B \wedge C)) \equiv ((A \vee B) \wedge (A \vee C))$.

```
(define (distOu F)
  (cond ((symbProp? F) ...)
        ((negFbf? F) ...)
        ((conjFbf? F)  ...)
         (else          ; c'est donc une disjonction
            (let ( (Fg (distOu (filsG F))) (Fd (distOu (filsD F))) )
                 (cond ((conjFbf? Fg) (list '^ (distOu (list 'v (filsG Fg) Fd)) (distOu (list 'v (filsD Fg) Fd))))
                       ((conjFbf? Fd) (list '^ (distOu (list 'v Fg (filsG Fd))) (distOu (list 'v Fg (filsD Fd)))))
                       (else  ; il n'y a plus de ^ dans les sous-formules
                                       (list 'v  Fg Fd)))))))
```

**Q 30** Écrire alors la fonction `formeConj` qui prend en paramètre une fbf quelconque et retourne une fbf logiquement équivalente sous forme conjonctive.

# 7 Forme clausale

On propose de représenter un littéral par un symbole ou la négation d'un symbole. Une clause sera représentée par la liste (sans doublon) des littéraux qui la composent. Et une forme clausale par la liste de ces clauses. Ci-suit des exemples de clause et forme clausale que vous pourrez utiliser pour tester vos fonctions :

```
(define exClause '( p (! r) t))
(define exFormeClausale '( (p (! p))   (p q (! r))   ((! r) s)   (p (! r) t)   (p ( ! r))   (r (! t))
                           (s t)   (p (! s))   ((! p) (! s))) )
```

On dispose par ailleurs des fonctions :

— `litteral?` : $PROP(S) \rightarrow Bool$ qui teste si une formule bien formée est un littéral ;

— `oppose` : $Litteral \rightarrow Litteral$ qui étant donné un littéral retourne son opposé.

Enfin, on va être amener à faire des ajouts et des unions dans des ensembles d'ensembles (ces derniers, les clauses, n'étant pas forcément ordonnés de la même manière). Il faut donc redéfinir les opérateurs `set-member?`, `set-add`, `set-union` pour les rendre indépendants de l'ordre des éléments. On fournit dont les fonctions suivantes (où `S` est un ensemble d'ensembles et `s` est un ensemble) :

— `(setSet-member? S s)` qui renvoie vrai si l'ensemble `s` est un élement de l'ensemble `S` ;

— `(setSet-add? S s)` qui retourne l'ensemble composé des éléments de `S` et de `s` (s'il n'est pas déjà dans `S`) ;

— `(setSet-union? S1 S2)` qui retourne l'ensemble composé des éléments de `S1` et des éléments de `S2`.

**Q 31** Écrire une fonction récursive `transClause` qui prend en paramètre une fbf disjonction de littéraux et retourne la clause correspondante à cette fbf.

**Q 32** Écrire une fonction récursive `transEnsClause` qui prend en paramètre une fbf sous forme conjonctive et retourne l'ensemble de clauses correspondant à cette fbf.

**Q 33** Finalement, écrire une fonction `formeClausale` qui prend en paramètre une fbf quelconque et retourne l'ensemble de clauses correspondant à sa forme clausale.

**Q 34** Ecrire une fonction `tautologique` qui teste si une clause passée en paramètre est valide.

**Q 35** Ecrire une fonction `subsume` qui teste si une clause subsume une autre.

**Q 36** Ecrire une fonction `simplifier` qui étant donnée une forme clausale passée en paramètre retourne la forme clausale obtenue en se débarassant des clauses tautologiques et subsumées.

# 8 Méthode de résolution

**Q 37** Ecrire une fonction `resolvable` qui prend deux clauses en paramètre et teste si elles possèdent un et un seul littéral opposé.

**Q 38** Ecrire une fonction `resolvante` qui prend deux clauses en paramètre ayant un et un seul littéral opposé et calcule la clause résolvante.

**Q 39** Ecrire une fonction `ClauseVideParResolution` qui étant donnée une forme clausale, retourne vrai si la clause vide est productible par résolution et faux sinon.

**Q 40** Ecrire des fonctions `satisfiableResol`, `valideResol` et `consequenceResol` qui s'appuie sur cette méthode.

# 9 Application

**Q 41** Modéliser le problème suivant en logique des propositions et résolvez-le à l'aide de la méthode de résolution :

*Un meurtre s'est produit au lieu dit "plateau de Cluedo", vous êtes chargé de diriger l'enquête. Vous disposez de 3 suspects, Dr Olive (`O`), Mlle Rose (`R`) et le Colonel Moutarde (`M`), avec 3 armes du crime potentielles, un chandelier (`H`), une clé anglaise (`A`), une corde (`D`) et 3 lieux possibles, la véranda (`V`), la cuisine (`C`) ou le bureau (`B`). Vous avez déjà réussi à récolter les indices suivants :*

1. *Il y a une et une seule arme du crime.*
2. *Il y a un et un seul lieu du crime.*
3. *Si le crime a eu lieu dans le bureau, la clé anglaise ne peut pas être l'arme du crime.*
4. *Si Mlle Rose est coupable, le crime a eu lieu dans la cuisine mais pas avec la corde.*
5. *Si le colonel Moutarde est l'assassin il n'a pas tué avec la clé anglaise.*
6. *Mlle Rose ou le colonel Moutarde est innocent (ou les deux).*
7. *Si Mlle Rose est innocente le crime a eu lieu dans le bureau ou avec la clé anglaise (ou les deux).*
8. *Le crime a eu lieu dans la véranda si et seulement si la clé anglaise est l'arme du crime.*
9. *Si le crime n'a pas eu lieu avec le chandelier ou si le docteur Olive est coupable (ou les deux) alors Mlle Rose est coupable aussi.*
10. *Le crime a eu lieu dans le bureau ou bien dans la cuisine.*
11. *Si le crime a eu lieu dans la cuisine alors l'arme du crime est la corde.*
12. *Enfin, vous êtes sûr qu'au moins un de vos trois suspects est coupable.*

*Deux de vos coéquipiers sont certains d'avoir trouvé la solution mais ils proposent des conclusions contradictoires :*

1. *Noémie affirme que c'est le colonel Moutarde dans le bureau avec le chandelier,*
2. *alors que Arthur est persuadé que ce sont le Dr Olive et Mlle Rose dans la cuisine avec la corde.*

*Est-ce que l'un des deux a raison ? Si oui, lequel ? ? ?*