LANGAGES DE PROGRAMMATION
IFT-3000
HIVER 2018

Travail pratique 2 - TRAVAIL INDIVIDUEL OU EN ÉQUIPE
Pondération de la note finale : 15%

À remettre, avant 23h59, le vendredi 27 avril 2018

1 Énoncé

Ce travail pratique consiste à compléter l'implantation d'un démonstrateur automatique de propositions. Ce démonstrateur est basé sur la méthode de *résolution*.

Dans ce qui suit, nous présentons la syntaxe des formules de la logique propositionnelle en utilisant respectivement la notation mathématique, la notation concrète utilisée dans le TP et la notation abstraite manipulée par les algorithmes du TP.

TABLE 1 – Notation mathématique.

P	::=	$1 \mid 0 \mid p \mid P \Leftrightarrow P' \mid P \Rightarrow P' \mid P \vee P' \mid P \wedge P' \mid \neg P \mid (P)$

TABLE 2 – Notation concrète.

P	::=	vrai $\mid$ faux $\mid id \mid P$ <=> $P' \mid P$ => $P' \mid P$ ou $P' \mid P$ et $P' \mid$ non $P \mid (P)$

TABLE 3 – Notation abstraite.

P	::=	Vrai $\mid$ Faux $\mid$ Var$(id) \mid$ Equ$(P, P') \mid$ Imp$(P, P') \mid$ Ou$(P, P') \mid$ Et$(P, P') \mid$ Non(P)

Nous utiliserons, dans cet énoncé du TP, la notation mathématique pour expliquer certaines propriétés. Lorsque vous allez tester votre TP, grâce à un analyseur lexical et syntaxique (parseur), vous pourrez utiliser la notation concrète pour définir des propositions à vérifier. Quant aux algorithmes du TP, ils utiliseront la notation abstraite. Ainsi, l'analyseur lexical et syntaxique qui vous est fourni avec le TP prend en entrée des formules saisies à l'aide de la notation concrète et retourne en résultat la représentation interne (ML) de ces formules, c'est-à-dire leur représentation en notation abstraite.

2 Démonstrateur automatique

Comme nous l'avons mentionné dans la section précédente, l'objectif du TP est d'implanter un module offrant un ensemble de fonctions permettant de déduire automatiquement la véracité d'une propriété.

En entrée, l'utilisateur spécifie le problème à résoudre sous forme d'hypothèses $H_1; H_2; \ldots; H_n$ et d'une conclusion C à prouver. Le problème consiste alors à prouver que :

$$H_1 \wedge H_2 \wedge \ldots \wedge H_n \Rightarrow C$$

c'est-à-dire que la conclusion C peut être déduite à partir de la conjonction d'hypothèses H_i.

2.1 Méthode de résolution

Pour atteindre l'objectif, on utilise, dans la littérature, des règles qui permettent, à partir des hypothèses, d'atteindre la conclusion. Par exemple, la règle suivante :

$$((p \Rightarrow q) \wedge p) \Rightarrow q$$

pourrait être utilisée pour déduire une propriété q à partir des hypothèses $p \Rightarrow q$ et p.

Dans le cadre de ce TP, nous allons utiliser le raisonnement par l'absurde pour déduire une conclusion C à partir d'hypothèses H_i. Autrement dit, pour démontrer que :

$$H_1 \wedge H_2 \wedge \dots H_n \Rightarrow C$$

nous allons supposé que C n'est pas vraie, c'est-à-dire que $\neg C$ est vraie, et montrer que ceci est absurde (faux). Ceci s'exprime comme suit :

$$H_1 \wedge H_2 \wedge \dots H_n \wedge \neg C \Rightarrow 0$$

(nous rappelons que 0 représente la proposition faux).

Afin de résoudre ce problème, nous allons utiliser (explicitement) qu'une seule règle :

TABLE 4 – Règle de résolution.

$(p \vee q) \wedge (\neg p \vee r) \Rightarrow (q \vee r)$

L'intérêt de cette règle réside dans la disparition d'une proposition, en l'occurrence p.

2.2 Exemple

Soient les propositions suivantes (cet exemple est présenté dans le fichier "test3.txt") :

n : « il neige »
p : « jean a revêtu sa polaire »
f : « jean a froid »

Soient les hypothèses suivantes :

$n \Rightarrow p$: « s'il neige, jean met sa polaire » (H_1)
$p \Rightarrow \neg f$: « si jean a sa polaire, il n'a pas froid » (H_2)
$\neg n \Rightarrow \neg f$: « s'il ne neige pas, jean n'a pas froid » (H_3)

On veut conclure que :

$\neg f$: « jean n'a jamais froid » (C)

Il s'agierait donc de montrer que

$H_1 \wedge H_2 \wedge H_3 \Rightarrow C$

Par raisonnement par l'absurde, il s'agiera de montrer que :

$H_1 \wedge H_2 \wedge H_3 \wedge \neg C \Rightarrow 0$

c'est-à-dire, en remplaçant les hypothèses et la conclusion par leur définition respective, que :

$(n \Rightarrow p) \wedge (p \Rightarrow \neg f) \wedge (\neg n \Rightarrow \neg f) \wedge \neg(\neg f) \Rightarrow 0$

Voici les étapes de la résolution (ces étapes résultent de l'utilisation de la fonction « decisionTrace ») :

$(n \Rightarrow p) \wedge (p \Rightarrow \neg f) \wedge (\neg n \Rightarrow \neg f) \wedge \neg(\neg f)$
$(1) \Rightarrow\ (\neg n \vee \underline{p}) \wedge (\underline{\neg p} \vee \neg f) \wedge (n \vee \neg f) \wedge f$
$(2) \Rightarrow\ (\underline{\neg n} \vee \neg f) \wedge (\underline{n} \vee \neg f) \wedge f$
$(3) \Rightarrow\ \underline{\neg f} \wedge \underline{f}$
$(4) \Rightarrow\ \underline{0}$

L'étape (1) est une simple traduction (rôle de la fonction « mfc ») : en effet, nous avons remplacé le symbole $\Rightarrow$ par le symbole $\vee$, et avons simplifié les propositions $\neg\neg f$ et $\neg\neg n$ en respectivement f et n, en utilisant, implicitement, les deux règles suivantes :

$$(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)$$
$$\neg\neg p \Leftrightarrow p$$

Dans les étapes (2), (3) et (4), nous avons utilisé la règle de résolution (les propositions qui disparaissent à chaque étape ont été soulignées). Aussi, à l'étape (3), nous avons enlevé les doublons ($\neg f$) en utilisant implicitement la règle (fonction « union » utilisée par la fonction « mfc ») :

$$p \wedge p \Leftrightarrow p$$

2.3 Démarche

Voici la démarche suivie dans ce TP :

1. En entrée, nous avons une liste d'hypothèses et une conclusion respectant la grammaire concrète présentée dans la table 2 (page 1). Les hypothèses sont séparées par le caractère « ; ». Il est possible d'utiliser des commentaires séparés par « (* » et « *) » Aussi, il est possible de saisir cette entrée sous forme de chaînes de caractères ou d'un fichier.

   En reprenant l'exemple précédent, supposons que l'on ait un fichier « test.txt » :

   ```
   (* Commentaires ... *)

   n => p ;
   p => (non f) ;
   (non n) => (non f) ;
   non f
   ```

2. À partir de l'entrée, on utilise un analyseur lexical et syntaxique, dont le code vous est fourni (fichier "parseur.ml"), pour transformer ces hypothèses et cette conclusion en notation abstraite (table 3), exploitable par les algorithmes du TP. L'analyseur syntaxique nous retourne une paire de valeurs dont la premier élément et une liste de propositions (correspondant à la liste des hypothèses) et le deuxième élément est une proposition (la conclusion).

   Plus précisément, cette analyseur retourne une valeur de type *enonce_probleme* défini comme suit :

   type *enonce_probleme* = *proposition list* * *proposition*

   Notons que *proposition* est un type défini comme suit :

   ```
   type proposition = Vrai
                    | Faux
                    | Non of proposition
                    | Et of proposition * proposition
                    | Ou of proposition * proposition
                    | Imp of proposition * proposition
                    | Equ of proposition * proposition
                    | Var of string ;
   ```

   ce qui correspond bien à la définition de la syntaxe abstraite (table 3).

   Par conséquent, en considérant notre exemple, l'analyseur syntaxique retourne la paire suivante :
   ([Imp (Var "n", Var "p") ; Imp (Var "p", Non (Var "f")) ;Imp (Non (Var "n"), Non (Var "f"))],Non (Var "f"))

3. Cet enoncé est transformé, grâce à une fonction « enonce2proposition » en une unique proposition. Le résultat obtenu est la valeur :
   Et (Imp (Var "n", Var "p"),Et (Imp (Var "p", Non (Var "f")),Et (Imp (Non (Var "n"), Non (Var "f")), Non (Non (Var "f")))))
   c'est-à-dire qu'on passe, en utilisant la notation concrète, de l'énoncé (la paire) :

   ([n => p, p => (non f), (non n) => (non f)],non f)

   à la proposition :

   (n => p) et (p => (non f)) et ((non n) => (non f)) et non(non f)

4. Par la suite, on transforme (fonction « mfc ») cette proposition en une forme particulière qui pourra être traitée par la méthode de résolution afin qu'elle soit réduite progressivement et aboutir éventuellement à la proposition faux.

2.4 Complément d'information à propos de la méthode de résolution

Dans la section 2.1, nous avons conclu que l'objectif était de prendre la proposition

$$H_1 \wedge H_2 \wedge \ldots H_n \wedge \neg C$$

et d'aboutir à 0 (faux), en utilisant la règle de résolution :

$$(\underline{p} \vee q) \wedge (\underline{\neg p} \vee r) \Rightarrow (q \vee r)$$

Comme nous pouvons le constater, pour pourvoir utiliser cette règle, il faut que notre proposition soit de la forme :

$$(p_1^1 \vee \ldots \vee p_1^k) \wedge \ldots \wedge (p_n^1 \vee \ldots \vee p_n^m)$$

c'est-à-dire des conjonctions (et) de disjonctions (ou).

2.4.1 Litteraux, Clauses et forme clausale

Un litteral est une variable propositionnelle (p) ou la négation d'une variable propositionnelle ($\neg p$).

Une clause est une proposition de la forme $l_1 \vee \ldots \vee l_m$ où chaque l_i est un litteral. Dans le TP, une clause est définie par le type suivant (à défaut de disposer d'un type *litteral*, on utilise le type *proposition* pour désigner la liste des littéraux qui forment une clause) :

type *clause* = *proposition list*

Une forme clausale est une proposition de la forme $c_1 \wedge \ldots \wedge c_s$ où chaque c_j est une clause. Dans le TP, une clause est définie par le type suivant :

type *forme_clausale* = *clause list*

Par conséquent, notre objectif sera de transformer une proposition, issue de l'énoncé (paire), en une forme clausale afin de pouvoir appliquer la règle de résolution : c'est le rôle de la fonction « mfc » que vous avez à implanter.

2.4.2 Construction d'une forme clausale

Soient $c_1 = a_1^1 \vee \ldots \vee a_1^n$ et $c_2 = a_2^1 \vee \ldots \vee a_2^m$ deux clauses. On note $c_1 \,\overline{\vee}\, c_2$ « l'union » de ces deux clauses c'est-à-dire :

$$c_1 \,\overline{\vee}\, c_2 = a_1^1 \vee \ldots \vee a_1^n \vee a_2^1 \vee \ldots \vee a_2^m$$

dans laquelle on a supprimé toute répétition de proposition a_i^j (règle implicite : $p \vee p \Leftrightarrow p$).

De même, étant données deux formes clausales fc_1 et fc_2, on note $fc_1 \,\overline{\wedge}\, fc_2$, leur « union », c'est-à-dire la conjonction de leurs clauses dans laquelle on a supprimé toute répétition (règle implicite : $p \wedge p \Leftrightarrow p$).

Comme les clauses et les formes clausales sont représentées par des listes, les opérateurs $\overline{\vee}$ et $\overline{\wedge}$ correspondent simplement à une fonction « union » que vous avez à implanter.

Soient $fc_1 = c_1 \wedge \ldots \wedge c_n$ et $fc_2 = c'_1 \wedge \ldots \wedge c'_m$ deux formes clausales. On note leur produit, noté $fc_1 \otimes fc_2$, la forme clausale définie par :

$$fc_1 \otimes fc_2 = \overline{\bigwedge}_{i,j}\, c_i \,\overline{\vee}\, c'_j$$

Exemple Soient $fc_1 = c_1 \wedge c_2$ et $fc_2 = c_3 \wedge c_4$, nous avons :

$$fc_1 \otimes fc_2 = (c_1 \,\overline{\vee}\, c_3) \,\overline{\wedge}\, (c_1 \,\overline{\vee}\, c_4) \,\overline{\wedge}\, (c_2 \,\overline{\vee}\, c_3) \,\overline{\wedge}\, (c_2 \,\overline{\vee}\, c_4)$$

L'opérateur $\otimes$ correspond à la fonction « prod » que vous avez à implanter.

Pour construire la forme clausale d'une proposition, on utilise la fonction suivante :

$$\begin{array}{lcl}
\mathcal{C}(1) & = & 1 \\
\mathcal{C}(0) & = & 0 \\
\mathcal{C}(p) & = & p \\
\mathcal{C}(\neg p) & = & \neg p \\
\mathcal{C}(\neg\neg P) & = & \mathcal{C}(P) \\
\mathcal{C}(\neg(P_1 \wedge P_2)) & = & \mathcal{C}(\neg P_1 \vee \neg P_2) \\
\mathcal{C}(\neg(P_1 \vee P_2)) & = & \mathcal{C}(\neg P_1 \wedge \neg P_2) \\
\mathcal{C}(\neg(P_1 \Rightarrow P_2)) & = & \mathcal{C}(\neg(\neg P_1 \vee P_2)) \\
\mathcal{C}(\neg(P_1 \Leftrightarrow P_2)) & = & \mathcal{C}(\neg P_1 \Leftrightarrow P_2) \\
\mathcal{C}(P_1 \wedge P_2) & = & \mathcal{C}(P_1) \,\overline{\wedge}\, \mathcal{C}(P_2) \\
\mathcal{C}(P_1 \vee P_2) & = & \mathcal{C}(P_1) \otimes \mathcal{C}(P_2) \\
\mathcal{C}(P_1 \Rightarrow P_2) & = & \mathcal{C}(\neg P_1 \vee P_2) \\
\mathcal{C}(P_1 \Leftrightarrow P_2) & = & \mathcal{C}((P_1 \Rightarrow P_2) \wedge (P_2 \Rightarrow P_1))
\end{array}$$

Cette fonction correspond à la fonction « mfc » que vous devez compléter.

2.4.3 Étapes de la résolution

Étant donné que notre énoncé a été transformé en une forme clausale représentée, dans le TP, par une liste de clauses, il reste à chercher deux clauses qui admettent des littéraux contraires (p et $\neg p$) pour pouvoir appliquer la règle de résolution et ainsi réduire ces deux clauses en une troisième.

Par conséquent, à partir de la listes des clauses, il faut considérer toutes les combinaisons de paires de clauses et explorer toutes les possibilités d'application de la règle de résolution. À cette fin, vous devez implanter une fonction « paires » qui a pour objectif de générer, à partir d'une liste, une liste de combinaisons de paires.

Par ailleurs, la fonction « resolutions », à implanter, prend 2 clauses et retourne la liste des clauses qui peuvent être déduites par application de la règle de résolution.

Finalement, la fonction « decision » prend en argument une proposition et décide si elle est vraie ou pas (retourne *true* si elle est vraie ; *false* sinon). De même, la fonction « decisisonTrace » retourne éventuellement (type *option*) la liste des formes clausales permettant d'aboutir à la preuve (par l'absurde) de la véracité d'une proposition donnée en entrée. Elle retourne None si la proposition n'est pas vraie.

3 Description des fonctions à implanter

Voici la liste minimale des fonctions que vous devez implanter :

- union (7 pts) : *'a list -> 'a list -> 'a list*
- prod (10 pts) : *'a list list -> 'a list list -> 'a list list*
- paires (15 pts) : *'a list -> (('a * 'a) * 'a list) list*
- enonce2proposition (8 pts) : *enonce_probleme -> proposition*
- mfc (10 pts) : *proposition -> forme_clausale*
- resolutions (20 pts) : *clause -> clause -> clause list*
- decision (20 pts) : *proposition -> bool*
- decisionTrace (10 pts) : *proposition -> forme_clausale list option*

Vous pouvez utiliser n'importe quel paradigme de programmation, ou en combiner. L'important est que le module « Resolution » implanté respecte la signature « RESOLUTION ».

Voici quelques informations au sujet des fonctions à implanter :

— La fonction « union » fait l'union de deux listes ; la liste résultante comprend les éléments des deux listes en argument (l'ordre des éléments n'importe pas), mais avec une seule occurrence de chaque élément :

```
# union [1;2;3;2;4;1] [2;1;5;3;6];;
- : int list = [1; 2; 3; 4; 5; 6]

# union ["a"] ["b"; "a"];;
- : string list = ["a"; "b"]

# union [Non (Var "a"); Var "b"] [Non (Var "b"); Var "a"; Var "b"];;
- : TypesUtiles.proposition list =
[Non (Var "a"); Var "b"; Non (Var "b"); Var "a"]
```

— La fonction « prod » implante l'opérateur $\otimes$ présenté dans la section 2.4.2 (page 4). Néanmoins, son type étant polymorphe, on peut l'appliquer à des liste de différents types :

```
# prod [[1;2];[3]] [[4];[5;2;3];[1]];;
- : int list list =
[[1; 2; 4]; [1; 2; 5; 3]; [1; 2]; [3; 4]; [3; 5; 2]; [3; 1]]

# prod [[1;2;1]] [[3];[]];;
- : int list list = [[1; 2; 3]; [1; 2]]

# prod [[1;2]] [];;
- : int list list = []

# prod [] [[4;5];[6]];;
- : int list list = []

# prod [[Non (Var "a"); Var "b"]; [Var "a"]] [[Var "b"]; [Var "a"]];;
- : TypesUtiles.proposition list list =
[[Non (Var "a"); Var "b"]; [Non (Var "a"); Var "b"; Var "a"];
[Var "a"; Var "b"]; [Var "a"]]
```

l'ordre des éléments de la liste résultante, ou des éléments des sous-listes résultantes, n'importe pas.

— La fonction « paires » prend en entrée une liste $l = [v_1, \ldots, v_n]$ et retourne la liste de tous les couples $((n_i, n_j), l \setminus \{n_i, n_j\})$. Voici quelques exemples d'utilisation de cette fonction dont le type est aussi polymorphe :

```
# paires [1;2;3;4];;
- : ((int * int) * int list) list =
[((1, 2), [3; 4]); ((1, 3), [2; 4]); ((1, 4), [2; 3]); ((2, 3), [1; 4]);
((2, 4), [1; 3]); ((3, 4), [1; 2])]

# paires ["a";"b";"c"];;
- : ((string * string) * string list) list =
[(("a", "b"), ["c"]); (("a", "c"), ["b"]); (("b", "c"), ["a"])]

# paires ["a";"b"];;
- : ((string * string) * string list) list = [(("a", "b"), [])]

# paires ["a"];;
- : ((string * string) * string list) list = []

# paires [];;
- : (('a * 'a) * 'a list) list = []

# paires [[Non (Var "a"); Var "b"]; [Non (Var "b")]; [Var "a"]];;
- : ((TypesUtiles.proposition list * TypesUtiles.proposition list) *
TypesUtiles.proposition list list) list
=
[(([Non (Var "a"); Var "b"], [Non (Var "b")]), [[Var "a"]]);
(([Non (Var "a"); Var "b"], [Var "a"]), [[Non (Var "b")]]);
(([Non (Var "b")], [Var "a"]), [[Non (Var "a"); Var "b"]])]
```

L'ordre des éléments en résultat n'importe pas.

— La fonction « enonce2proposition » prend en entrée un énoncé, tel que défini dans le TP via le type *enonce* (liste d'hypothèses et une conclusion), et retourne une proposition correspondant à la conjonction des hypothèses et de la négation de la conclusion :

```
# enonce2proposition ([Imp (Var "a", Var "b"); Non (Var "b")], Non (Var "a"));;
- : TypesUtiles.proposition =
Et (Imp (Var "a", Var "b"), Et (Non (Var "b"), Non (Non (Var "a"))))

# enonce2proposition ([Et (Var "a", Var "b")], Var "a");;
- : TypesUtiles.proposition = Et (Et (Var "a", Var "b"), Non (Var "a"))

# enonce2proposition ([Ou (Var "a", Var "b")], Et (Var "a", Non (Var "b")));;
- : TypesUtiles.proposition =
Et (Ou (Var "a", Var "b"), Non (Et (Var "a", Non (Var "b"))))

# enonce2proposition ([Var "a"], Var "a");;
- : TypesUtiles.proposition = Et (Var "a", Non (Var "a"))
```

— La fonction « mfc » est une implantation directe et naturelle de la fonction $\mathcal{C}$. Notez qu'étant donnée que son type de retour est une valeur de type *forme_clausale*, que cette dernière est définie comme une liste de clauses, elles-mêmes définies comme des listes de littéraux (variables propositionnelles ou négation de variables propositionnelles), la fonction retournera au final une liste de listes :

```
# mfc Vrai;;
- : TypesUtiles.proposition list list = []

# mfc Faux;;
- : TypesUtiles.proposition list list = [[]]

# mfc (Var "a");;
- : TypesUtiles.proposition list list = [[Var "a"]]

# mfc (Non Vrai);;
- : TypesUtiles.proposition list list = [[]]

# mfc (Non Faux);;
- : TypesUtiles.proposition list list = []

# mfc (Non (Var "a"));;
- : TypesUtiles.proposition list list = [[Non (Var "a")]]

# mfc (Non (Non (Var "a")));;
- : TypesUtiles.proposition list list = [[Var "a"]]

# mfc (Et (Ou (Var "a", Var "b"), Non (Et (Var "a", Non (Var "b")))));;
- : TypesUtiles.proposition list list =
[[Var "a"; Var "b"]; [Non (Var "a"); Var "b"]]
```

— La fonction « resolutions » prend en argument deux clauses et retourne la liste des clauses que l'on peut déduire par application de la règle de résolution. Par exemple, et en utilisant la notation mathématique, si on considère les deux clauses suivantes :

$c_1 = p \vee \neg q \vee r \vee \neg s$
$c_1 = r \vee \neg p \vee q \vee t$

La fonction « resolutions » devrait alors retourner la liste suivante :

$[\neg q \vee r \vee \neg s \vee q \vee t; p \vee r \vee \neg s \vee \neg p \vee t]$

Le premier élément de la liste est déduit de c_1 et c_2 en faisant disparaître p et $\neg p$. Le deuxième élément est déduit en faisant disparaître q et $\neg q$. Notons que les occurrences multiples de r ont été supprimées (grâce à l'application de la fonction « union »).

```
# let c1 = [Var "p"; Non(Var "q"); Var "r"; Non(Var "s")];;
val c1 : TypesUtiles.proposition list =
[Var "p"; Non (Var "q"); Var "r"; Non (Var "s")]

# let c2 = [Var "r"; Non(Var "p"); Var "q"; Var "t"];;
val c2 : TypesUtiles.proposition list =
[Var "r"; Non (Var "p"); Var "q"; Var "t"]

# resolutions c1 c2;;
- : TypesUtiles.proposition list list =
[[Non (Var "q"); Var "r"; Non (Var "s"); Var "q"; Var "t"];
[Var "p"; Var "r"; Non (Var "s"); Non (Var "p"); Var "t"]]

# resolutions [Non (Var "a"); Var "b"] [Non (Var "b")];;
- : TypesUtiles.proposition list list = [[Non (Var "a")]]

# resolutions [Non (Var "a"); Var "b"] [Var "a"];;
- : TypesUtiles.proposition list list = [[Var "b"]]

# resolutions [Non (Var "b")] [Var "a"];;
- : TypesUtiles.proposition list list = []

# resolutions [Non (Var "a"); Var "b"] [Non (Var "b"); Var "a"];;
- : TypesUtiles.proposition list list =
[[Var "b"; Non (Var "b")]; [Non (Var "a"); Var "a"]]

# resolutions [Var "b"] [Non (Var "b")];;
- : TypesUtiles.proposition list list = [[]]
```

— La fonction « decision » retourne *true* si une proposition passée en argument est vraie ; *false*, sinon. La fonction peut avantagement faire appel à la fonction « paires » pour déterminer toutes les paires à considérer et leur appliquer la fonction « resolutions » ; évidemment, elle doit considérer toutes les possibilités et rechercher si dans l'une d'entre elles, on aboutit au résultat escompté, soit l'apparition de la proposition Faux (représentée par une liste vide) dans une forme clausale (une forme clausale étant une conjonction de clauses ; si la proposition Faux apparait dans cette conjonction, toute cette dernière est alors fausse (règle $p \wedge 0 \Rightarrow 0$), ce qui correspondrait au résultat recherché puisqu'on utilise la preuve par l'absurde) :

```
# decision (enonce2proposition (aSynt (aLex "a␣=>␣b;␣non␣b;␣non␣a")));;
- : bool = true

# decision (enonce2proposition (aSynt (aLex "a␣=>␣b;␣non␣b;␣non␣b")));;
- : bool = true

# decision (enonce2proposition (aSynt (aLex "a␣=>␣b;␣non␣b;␣a")));;
- : bool = false

# decision (enonce2proposition (aSynt (aLex "p␣<=>␣q;␣q␣<=>␣r;␣p␣<=>␣r")));;
- : bool = true

# decision (enonce2proposition (aSynt (aLex "a␣=>␣b;␣a;␣b")));;
- : bool = true

# decision (Et (Imp (Var "a", Var "b"), Et (Var "a", Non (Var "b"))));;
- : bool = true
```

Notez qu'avant de faire appel à la fonction «paires», la fonction «decision» enlève de la forme clausale à traiter toutes les clauses qui sont vraies, c'est-à-dire celles qui comprennent à la fois des littéraux et leur négation ; en effet, puisque $p \vee \neg p \Rightarrow 1$, que $1 \vee p \Rightarrow 1$, et que $1 \wedge p \Rightarrow p$, par conséquent $(p \vee \neg p \vee \ldots \vee p') \wedge q \Rightarrow q$.
Cette remarque concerne aussi la fonction « decisionTrace ».

— La fonction « decisionTrace » retourne *None* si une proposition passée en argument n'est pas vraie ; elle retourne *Some v*, si elle est vraie. Dans ce dernier cas, la valeur *v* correspond à la liste des formes clausales qui ont été utilisées afin d'aboutir à la conclusion :

```
# decisionTrace (enonce2proposition (aSynt (aLex "a => b; non b; non a")));;
- : TypesUtiles.forme_clausale list option =
Some
[[[Non (Var "a"); Var "b"]; [Non (Var "b")]; [Var "a"]];
[[Non (Var "a")]; [Var "a"]]; [[]]]

# decisionTrace (enonce2proposition (aSynt (aLex "a => b; non b; a")));;
- : TypesUtiles.forme_clausale list option = None

# decisionTrace (enonce2proposition (aSynt (aLex "p <=> q; q <=> r; p <=> r")));;
- : TypesUtiles.forme_clausale list option =
Some
[[[Non (Var "p"); Var "q"]; [Non (Var "q"); Var "p"];
[Non (Var "q"); Var "r"]; [Non (Var "r"); Var "q"]; [Var "p"; Var "r"];
[Non (Var "r"); Non (Var "p")]];
[[Non (Var "p"); Var "r"]; [Non (Var "q"); Var "p"];
[Non (Var "r"); Var "q"]; [Var "p"; Var "r"];
[Non (Var "r"); Non (Var "p")]];
[[Var "r"]; [Non (Var "q"); Var "p"]; [Non (Var "r"); Var "q"];
[Non (Var "r"); Non (Var "p")]];
[[Var "p"; Non (Var "r")]; [Var "r"]; [Non (Var "r"); Non (Var "p")]];
[[Non (Var "r")]; [Var "r"]]; [[]]]

# decisionTrace (enonce2proposition (aSynt (aLex "a => b; a; b")));;
- : TypesUtiles.forme_clausale list option =
Some
[[[Non (Var "a"); Var "b"]; [Var "a"]; [Non (Var "b")]];
[[Var "b"]; [Non (Var "b")]]; [[]]]
```

Notez que dans la plupart des fonctions à implanter, l'ordre des éléments des listes retournées n'importe pas.

4 Code source

Dans ce TP, nous vous fournissons le code source de l'analyseur lexical et syntaxique ainsi que celui de certaines fonctions exploitant le TP (fichier tp2.ml). Plus précisément, voici les fichiers qui sont fournis :

— typesutiles.ml : principales structures de données utilisées par le parseur et le TP.
— parseur.ml : analyseur lexicale et syntaxique (des exemples sont fournis en commentaire à la fin du fichier).
— resolution.mli : signature des fonctions du TP à implanter.
— resolution.ml : module du TP à implanter (des exemples sont fournis en commentaire à la fin du fichier).
— tp2.ml : comprend des fonctions permettant d'exploiter aisément les principales fonctionnalités du projet que constitue le travail pratique (des exemples sont fournis en commentaire à la fin du fichier).
— test?.txt : 4 fichiers tests.
— header.html et footer.html : fichiers utilisés par tp2.ml afin de produire un résultat visualisé dans un navigateur.

Une fois votre TP complété, vous pouvez l'exploiter en utilisant des fonctions définies dans le fichier tp2.ml :

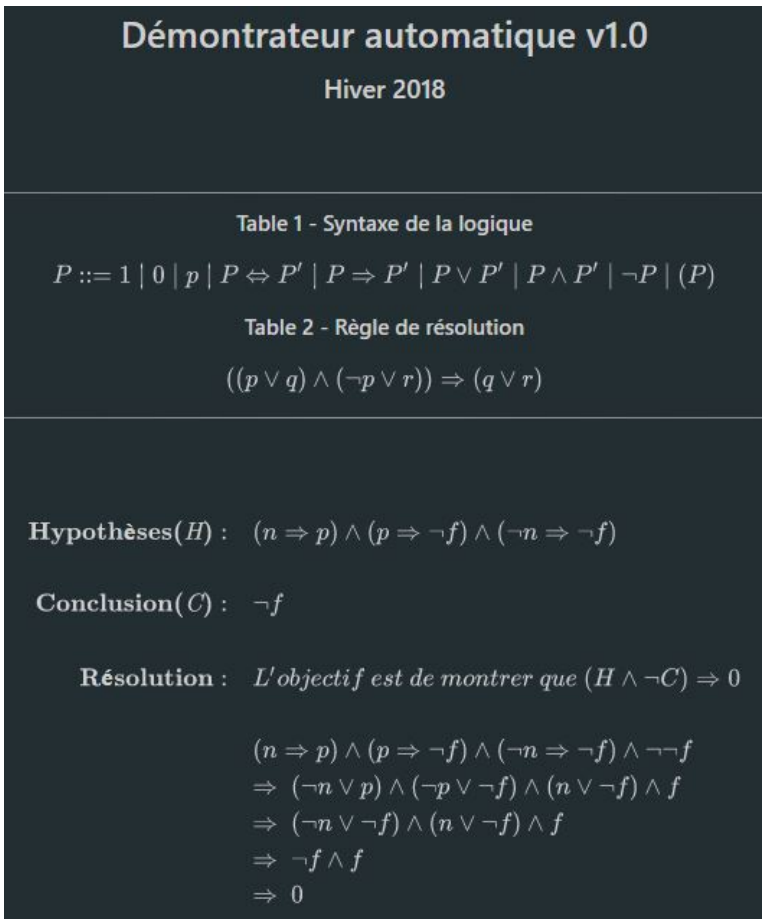
```
# tp2Trace ~format:Texte "n => p; p => (non f); (non n) => (non f); non f";;
Hypotheses (H):
    (n => p) et (p => (non f)) et ((non n) => (non f))

Conclusion (C):
    (non f)

Resolution: L'objectif est de montrer que (H et (non C)) => faux

    (n => p) et (p => (non f)) et ((non n) => (non f)) et (non (non f))
    => ((non n) ou p) et ((non p) ou (non f)) et (n ou (non f)) et f
    => ((non n) ou (non f)) et (n ou (non f)) et f
    => (non f) et f
    => faux
- : unit = ()

# tp2Trace "n => p; p => (non f); (non n) => (non f); non f";;
Voir affichage dans navigateur.
- : unit = ()
```



5 À remettre

Il faut remettre un fichier .zip contenant uniquement le fichier "resolution.ml" complété. Le code doit être clair et bien structuré [1].

6 Remarques importantes

Plagiat : Tel que décrit dans le plan de cours, le plagiat est interdit. Une politique stricte de tolérance zéro est appliquée en tout temps et sous toutes circonstances. Tous les cas seront référés à la direction de la Faculté.

Travail remis en retard : Tout travail remis en retard se verra pénalisé de 25% par jour de retard. Chaque journée de retard débute dès la limite de remise dépassée (dès la première minute). Un retard excédant 2 jours (48h) provoquera le rejet du travail pour la correction et la note de 0 pour ce travail ; de même pour un fichier .zip qui ne comprend pas le fichier du TP (comprendrait des fichiers de travaux d'autres cours) : rejet du travail pour la correction. La remise doit se faire par la boite de dépôt du TP2 dans la section « Évaluation et résultats ».

Barème : Au niveau des fonctions à implanter, le barème est précisé dans le fichier "resolution.mli". Notez cependant que :

— (-15pts), si votre code ne compile pas (provoque une erreur/exception lors du chargement du fichier "resolution.ml" dans l'interpréteur, i.e. « **#use** "resolution.ml";; ») ; évidemment, seule le corps du module « Resolution » peut être complété et modifié (toute la partie qui vous est déjà fournie ne peut être modifiée, en particulier le nom du module, « Resolution », et la restriction qu'il doit vérifier la signature « RESOLUTION ») ;

— (-5pts), si le fichier .zip retourné comprend autre chose que le fichier "resolution.ml", ou que ce dernier a été renommé ;

Bon travail.

1. Suggestion : http://ocaml.org/learn/tutorials/guidelines.html