

Dans les pas d'Indi

OPTION INFORMATIQUE - Devoir n° 3 - Olivier Reynet

Toutes les réponses informatiques à cet examen sont à fournir en langage OCaml.

A Logique des propositions (CCINP 2003)

Dans l'Égypte ancienne, la protection de la tombe des Pharaons faisait l'objet d'une grande attention. Le sarcophage était porté dans la salle funéraire par des esclaves accompagnés d'un prêtre. Le couloir y conduisant était définitivement condamné après leur passage. Les esclaves étaient ensuite abandonnés dans la salle funéraire et le prêtre quittait le tombeau par un passage secret. Pour éviter que les esclaves et les éventuels pillards puissent emprunter le même passage, celui-ci était fermé par plusieurs portes dont l'ouverture demandait la réponse à des énigmes. Pour empêcher qu'une autre personne que le prêtre ne puisse trouver la réponse aux énigmes, celles-ci étaient régies par des règles propres à chaque tombe et connues uniquement du prêtre et de son ordre.

Vous faites partie d'une équipe d'archéologues qui explore un tombeau récemment découvert grâce au déchiffrement d'un manuscrit écrit en hiéroglyphes contenant l'emplacement du tombeau et la précieuse règle nécessaire à la résolution des énigmes :

«Chaque énigme est composée de trois affirmations. Une affirmation parmi les trois est toujours fausse, les deux autres sont toujours vraies. Attention, ce ne sont pas forcément toujours les mêmes.»

Votre équipe a réussi à déblayer l'entrée de la salle funéraire et vous vous y êtes précipités trop imprudemment. En effet, la galerie, mal étayée, s'est effondrée après votre passage. Vous ne disposez pas de suffisamment d'air pour attendre que vos collègues dégagent à nouveau le passage. Votre seule chance de survie est d'emprunter le passage secret usuellement réservé au prêtre.

Nous noterons A_1 , A_2 et A_3 les propositions associées aux trois affirmations contenues dans les énigmes apparaissant sur chaque porte.

A1. Représenter la règle sous la forme d'une formule du calcul des propositions dépendant de A_1 , A_2 et A_3 .

Solution : La règle «une seule des affirmations est vraie» se traduit par la formule :

$$(\neg A_1 \wedge A_2 \wedge A_3) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (A_1 \wedge A_2 \wedge \neg A_3)$$

Une première porte bloque le passage. Devant cette porte se trouvent une dalle blanche et une dalle noire. Les affirmations suivantes sont inscrites sur la porte :

- A_1 : Si tu poses le pied sur la dalle blanche, alors pose le pied sur la dalle noire!
- A_2 : Pose les pieds simultanément sur les dalles blanche et noire!
- A_3 : Pose le pied sur la dalle noire!

Nous noterons B, respectivement N, les variables propositionnelles correspondant au fait de poser le pied sur la dalle blanche, respectivement noire.

A2. Exprimer A_1 , A_2 et A_3 sous la forme de formules du calcul des propositions dépendant de B et de N.

Solution :

$$A_1 \equiv B \Rightarrow N \equiv \neg B \vee N$$

$$A_2 \equiv B \wedge N$$

$$A_3 \equiv N$$

A3. En utilisant le calcul des propositions (résolution avec les formules de De Morgan), déterminer la (ou les) dalle(s) sur laquelle (ou lesquelles) vous devez poser les pieds pour ouvrir cette porte.

Solution :

$$\neg A_1 \wedge A_2 \wedge A_3 \equiv (B \wedge \neg N) \wedge (B \wedge N) \wedge N \equiv B \wedge \neg N \wedge N$$

ce qui est une contradiction, c'est-à-dire toujours faux à cause de $\neg N \wedge N$.

$$A_1 \wedge A_2 \wedge \neg A_3 \equiv (\neg B \vee N) \wedge (B \wedge N) \wedge \neg N \equiv (\neg B \vee N) \wedge B \wedge N \wedge \neg N$$

ce qui est aussi une contradiction.

La règle est donc équivalente :

$$A_1 \wedge \neg A_2 \wedge A_3 \equiv (\neg B \vee N) \wedge (\neg B \vee \neg N) \wedge N \equiv (\neg B \vee (N \wedge \neg N)) \wedge N = \neg B \vee N$$

par application de la distributivité de \vee par rapport à \wedge . Il faut donc poser le pied sur la dalle noire, voire les deux!

La porte s'ouvre, puis se referme après votre passage. Une seconde porte se trouve maintenant face à vous. Sur le côté de la porte se trouvent trois pavés de tailles différentes (petit, moyen, gros). Trois affirmations sont inscrites sur la porte. Malheureusement, un des hiéroglyphes est illisible.

Voici les textes que vous arrivez à déchiffrer :

- A_1 : L'affirmation suivante est fausse : n'appuie pas sur le petit pavé ou appuie sur le gros pavé!
- A_2 : N'appuie pas sur le moyen pavé mais appuie sur le gros pavé!
- A_3 : N'appuie ni sur le gros pavé, ni sur le ... pavé!

Les hiéroglyphes vous permettent de savoir que ... correspond soit à gros, soit à moyen, soit à petit. **Nous noterons P, M et G les variables propositionnelles correspondant au fait d'appuyer sur le petit, le moyen ou le gros pavé.**

A4. Exprimer A_1 , A_2 et A_3 sous la forme de formules du calcul des propositions dépendant de P, M et G.

Solution :

$$A_1 \equiv \neg(P \Rightarrow G) \equiv \neg(\neg P \vee G) \equiv P \wedge \neg G$$

$$A_2 \equiv \neg M \wedge G$$

$$A_3 \equiv \neg G \wedge (\neg P \vee \neg G \vee \neg M) \equiv \neg G \wedge (\neg P \vee \neg M)$$

- A5.** En utilisant le calcul des propositions (résolution avec les tables de vérité), déterminer le (ou les) pavé(s) sur lequel (ou lesquels) vous devez appuyer pour ouvrir cette porte.

Solution : On fait une table de vérité :

P	M	G	$P \wedge \neg G$	$\neg M \wedge G$	$\neg G \wedge \neg P$	$\neg G \wedge \neg M$	$\neg G$
0	0	0	0	0	1	1	1
0	0	1	0	1	0	0	0
0	1	0	0	0	1	0	1
0	1	1	0	0	0	0	0
1	0	0	1	0	0	1	1
1	0	1	0	1	0	0	0
1	1	0	1	0	0	0	1
1	1	1	0	0	0	0	0

Seule la cinquième ligne correspondant à $P = 1, M = 0, G = 0$ engendre deux affirmations vraies sur les trois. Le symbole caché ne peut être que M . Il faut donc appuyer sur le petit pavé Indi!

B Tautologie, équivalence et évaluation

- B6.** Soient a, b, c des formules. Montrer que les formules suivantes sont des tautologies :

- (a) $a \Rightarrow (b \Rightarrow a)$

Solution : On peut procéder par table de vérité :

a	b	$b \Rightarrow a$	F
0	0	1	1
0	1	1	1
1	0	0	1
1	1	1	1

ou par calcul :

$$a \Rightarrow (b \Rightarrow a) \equiv \neg a \vee \neg b \vee a \equiv \top$$

- (b) $(a \Rightarrow b) \vee (b \Rightarrow c)$

Solution : Par calcul :

$$(a \Rightarrow b) \vee (b \Rightarrow c) \equiv \neg a \vee b \vee \neg b \vee c \equiv \top$$

- B7.** Démontrer les équivalences suivantes ou trouver un contre-exemple :

- (a) $(a \Rightarrow b) \Rightarrow c \iff a \Rightarrow (b \Rightarrow c)$

Solution : La valuation $a = 0, b = 1, c = 0$ est un contre-exemple. Elle montre que cette équivalence n'est pas vraie.

(b) $(a \wedge b) \Rightarrow c \iff a \Rightarrow (b \Rightarrow c)$

Solution : Par le calcul on montre l'équivalence :

$$(a \wedge b) \Rightarrow c \equiv \neg(a \wedge b) \vee c \equiv \neg a \vee \neg b \vee c$$

$$a \Rightarrow (b \Rightarrow c) \equiv \neg a \vee \neg b \vee c$$

On dispose du type amgébrique récursif OCaml suivant :

```
1 type formule =
2   | T (* true *)
3   | F (* false *)
4   | Var of int (* variable *)
5   | Not of formule (* negation *)
6   | And of formule * formule (* conjonction *)
7   | Or of formule * formule (* disjonction *)
8   | Imp of formule * formule (* implication *)
9   | Equiv of formule * formule (* equivalence *)
```

Les variables logiques sont toujours numérotées par des entiers de 0 à $n - 1$ si la formule possède n variables.

- B8.** À l'aide du type `formule` ci-dessus, donner l'expression OCaml de la formule $\phi = (a \wedge b) \Rightarrow c \iff a \Rightarrow (b \Rightarrow c)$.

Solution :

```
1 let phi = Equiv(Imp(And(Var 0, Var 1), Var 2), Imp(Var 0, Imp(Var 1, Var 2)))
```

- B9.** Écrire une fonction récursive de signature `nb_var : formule -> int` qui renvoie le nombre de variables d'une formule logique. Utiliser une fonction interne auxiliaire récursive qui détermine l'indice le plus grand d'une variable dans la formule.

Solution :

```
1 let nb_var f =
2   let rec max_ind g k =
3     match g with
4       | T | F -> k
5       | Var i -> max i k
6       | Not fa -> max_ind fa k
7       | And (fa,fb) | Or (fa,fb) | Imp (fa,fb) | Equiv (fa, fb) -> max_ind fb (
8         max_ind fa k)
9   in (max_ind f 0) + 1;;
```

On cherche à vérifier qu'une formule logique est une tautologie en l'évaluant pour toutes les valuations possibles.

Pour évaluer des formules logiques, on choisit de représenter les valuations possibles des variables par des entiers. Chaque bit de cet entier représente une variable de la formule logique. Le bit de poids faible (0) représente la valuation de `var 0`, le second celle de `var 1` et ainsi de suite.

Par exemple, si la formule possède trois variables x_0, x_1 et x_2 ,

- l'entier $5_{10} = 101_2$ correspond à la valuation $x_2 = 1, x_1 = 0, x_0 = 1$,
- l'entier $3_{10} = 011_2$ correspond à la valuation $x_2 = 0, x_1 = 1, x_0 = 1$.

B10. Écrire une fonction de signature `get_v : int → int → bool` dont les paramètres sont une valuation et l'indice d'une variable. Cette fonction renvoie la valeur de vérité d'une variable d'après une valuation. Par exemple, `get_v 5 1` renvoie `false` et `get_v 3 1` renvoie `true`. Pour cette fonction, on utilisera les opérateurs OCaml :

- `land` : ET bit à bit sur deux entiers. Par exemple, `5 land 2` renvoie 0 et `5 land 4` renvoie 4. En effet : 101_2 ET $010_2 = 000_2$ et 101_2 ET $100_2 = 100_2 = 4$.
- `lsl` : décalage à gauche d'un entier. Elle permet de rapidement calculer une puissance de deux. Par exemple : `1 lsl 3` vaut 8, car $1000_2 = 2^3 = 8$.

Solution :

```
1 let get_v v k = (v land (1 lsl k)) != 0;;
```

B11. Écrire une fonction de signature `evaluation : int → formule → bool` qui évalue une formule logique d'après une valuation donnée par un entier.

Solution :

```
1 let rec evaluation v f =
2     match f with
3     | T → true
4     | F → false
5     | Var k → get_v v k
6     | Not p → not (evaluation v p)
7     | And (p, q) → evaluation v p && evaluation v q
8     | Or (p, q) → evaluation v p || evaluation v q
9     | Imp (p, q) → not (evaluation v p) || (evaluation v q)
10    | Equiv (p, q) → evaluation v p = evaluation v q;;
```

B12. Écrire une fonction de signature `est_tautologie : formule → bool` qui détermine par la force brute si une formule logique est une tautologie. On utilisera un style impératif avec une boucle `while`.

Solution :

```
1 let est_tautologie f =
2     let n = nb_var f in
3     let v_limit = 1 lsl n in
4     let v = ref 0 in
5     let taut = ref true in
6     while !v < v_limit && !taut do
7         if not (evaluation !v f) then taut := false;
8         incr v;
9     done;
10    !taut;;
```

C Système complet logique

Le tableau 1 définit les opérateurs NAND (\uparrow), NOR (\downarrow), XOR (\oplus) par leurs tables de vérité.

a	b	$a \uparrow b$	$a \downarrow b$	$a \oplus b$
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

TABLE 1 – Définition des opérateurs NAND, NOR et XOR

■ **Définition 1 — Complétude d'un ensemble d'opérateurs logiques.** Un ensemble S d'opérateurs logiques est complet si toute formule logique est équivalente à une formule qui n'utilise que des opérateurs de S .

(R) L'ensemble $S = \{\wedge, \vee, \neg\}$ est complet.

C13. Exprimer NAND (\uparrow), NOR (\downarrow), XOR (\oplus) à l'aide de \vee , \wedge et \neg .

Solution :

$$a \uparrow b = \neg(a \wedge b)$$

$$a \downarrow b = \neg(a \vee b)$$

$$a \oplus b = \neg a \wedge b \vee a \wedge \neg b$$

C14. Montrer que :

(a) $\neg a = a \uparrow a$

Solution : D'après la définition de \uparrow , on a :

$$a \uparrow b = \neg(a \wedge b)$$

On peut remarquer, d'après la table de vérité que :

$$a \uparrow b = \neg(a \wedge b) = \neg(b \wedge a)$$

c'est-à-dire \uparrow est commutative. Par contre, \uparrow n'est pas associative.

On peut donc en déduire que :

$$\neg a = \neg a \vee \neg a = \neg(a \wedge a) = a \uparrow a$$

(b) $a \wedge b = (a \uparrow b) \uparrow (a \uparrow b)$

Solution :

$$a \wedge b = \neg(a \uparrow b) = (a \uparrow b) \uparrow (a \uparrow b)$$

(c) $a \vee b = (a \uparrow a) \uparrow (b \uparrow b)$

Solution :

$$a \vee b = \neg(\neg a \wedge \neg b) = \neg a \uparrow \neg b = (a \uparrow a) \uparrow (b \uparrow b)$$

C15. Montrer par induction structurale sur les formules logiques que $\{\uparrow\}$ est complet.

Solution : Considérons la propriété \mathcal{P} «Le système $\{\uparrow\}$ est complet. »

(Cas de base) Si la formule logique est réduite à \top, \perp ou une variable x , il n'est besoin d'aucun constructeur pour les construire. \mathcal{P} est donc vérifiée.

(Pas d'induction) On suppose (hypothèse d'induction) qu'on dispose de formules ψ et ξ qui vérifient la propriété.

$(\neg\psi)$ D'après la question précédente, $\neg\psi = \psi \uparrow \psi$. Donc cette négation ne s'exprime qu'avec des \uparrow d'après notre hypothèse d'induction.

$(\psi \wedge \xi)$ D'après la question précédente, $\psi \wedge \xi = (\psi \uparrow \xi) \uparrow (\psi \uparrow \xi)$. Idem.

$(\psi \vee \xi)$ D'après la question précédente, $\psi \vee \xi = (\psi \uparrow \psi) \uparrow (\xi \uparrow \xi)$. Idem.

(Conclusion) Comme la propriété est vraie pour le cas de base et qu'elle est invariante par l'opération des constructeurs des formules logiques, elle est vraie pour toute formule φ exprimée avec des \uparrow .

$\{\uparrow\}$ est donc un ensemble complet.

C16. Montre que :

(a) $\neg a = a \downarrow a$

Solution : D'après la définition de \downarrow , on a :

$$a \downarrow b = \neg(a \vee b)$$

On peut remarquer, d'après la table de vérité que :

$$a \downarrow b = \neg(a \vee b) = \neg(b \vee a)$$

c'est-à-dire \downarrow est commutative. Par contre, \downarrow n'est pas associative.

On peut donc en déduire que :

$$\neg a = \neg a \wedge \neg a = \neg(a \vee a) = a \downarrow a$$

(b) $a \wedge b = (a \downarrow a) \downarrow (b \downarrow b)$

Solution :

$$a \wedge b = \neg(\neg a \vee \neg b) = \neg a \downarrow \neg b = (a \downarrow a) \downarrow (b \downarrow b)$$

(c) $a \vee b = (a \downarrow b) \downarrow (a \downarrow b)$

Solution :

$$a \vee b = \neg(a \downarrow b) = (a \downarrow b) \downarrow (a \downarrow b)$$

(d) Que peut-on en déduire?

Solution : L'ensemble $\{\downarrow\}$ est complet.

C17. On souhaite montrer que $\{\oplus\}$ n'est pas complet.

(a) Soit d , la valuation qui donne la valeur 0 à toutes les variables d'une formule logique. Montrer par induction structurale sur les formules logiques que pour une formule φ exprimée uniquement avec des \oplus et des variables propositionnelles, $\llbracket \varphi \rrbracket_d = 0$.

Solution :

(Cas de base) Si $\varphi = x$ est réduite à une seule variable, alors sa valuation par d est \perp .

(Pas d'induction) On suppose qu'on dispose de formules ψ et ξ qui vérifient la propriété et qui ne sont exprimées qu'avec des variables propositionnelles. Posons $\varphi = \xi \oplus \psi$. Mais alors $\llbracket \varphi \rrbracket_d = \llbracket \xi \oplus \psi \rrbracket_d$. Or, on sait que $\llbracket \xi \rrbracket_d = 0$ et $\llbracket \psi \rrbracket_d = 0$. Donc, d'après la table de vérité de l'opérateur \oplus , $\llbracket \varphi \rrbracket_d = 0$.

(Conclusion) Comme la propriété est vraie pour le cas de base et qu'elle est invariante par construction, elle est vraie pour toute formule φ exprimée avec des \oplus et des variables propositionnelles.

(b) En déduire que $\{\oplus\}$ n'est pas complet.

Solution : On remarque que $\xi = \neg a$ est une formule qui est vraie lorsque a vaut 0. On nomme d cette valuation de ξ . On procède par l'absurde.

Supposons que ξ puisse s'exprimer uniquement avec des \oplus . Alors, d'après la propriété précédente, on a nécessairement $\llbracket \xi \rrbracket_d = 0$. Ce qui est absurde puisqu'alors ξ est vraie.

On dispose des types OCaml suivants :

```

1  type formule =
2    | T (* true *)
3    | F (* false *)
4    | Var of int (* variable *)
5    | Not of formule (* negation *)
6    | And of formule * formule (* conjonction *)
7    | Or of formule * formule (* disjonction *)
8
9  type nand_formule =
10   | NT (* true *)
11   | NF (*false*)

```



```

12      | NVar of int (* x_k *)
13      | N of nand_formule * nand_formule

```

C18. Écrire une fonction de signature `std_to_nand : formule → nand_formule` qui transforme une formule exprimée dans le système standard $\{\neg, \wedge, \vee\}$ en une formule exprimée avec des \uparrow .

Solution :

```

1  let rec std_to_nand f =
2      match f with
3      | T → NT
4      | F → NF
5      | Var i → NVar i
6      | Not g → let ng = std_to_nand g in N(ng, ng)
7      | And(a,b) → let na = std_to_nand a and nb = std_to_nand b in N(N(na,nb), N(na,nb))
8      | Or(a,b) → let na = std_to_nand a and nb = std_to_nand b in N(N(na,na), N(nb,nb));;

```

C19. Écrire une fonction de signature `nand_to_std : nand_formule → formule` qui transforme une formule exprimée avec des \uparrow en une formule exprimée avec le système standard $\{\neg, \wedge, \vee\}$.

Solution :

```

1  let rec nand_to_std nf =
2      match nf with
3      | NT → T
4      | NF → F
5      | NVar i → Var i
6      | N(sfa, sfb) → Not(And(nand_to_std sfa, nand_to_std sfb));;

```

(R) La porte NAND est très utilisée en électronique : de nombreux circuits sont élaborés à l'aide de cette unique porte plus petite et qui consomme moins de puissance. On peut également facilement la transformer en porte non en reliant les deux entrées (cf. figure 1).

D Formes normales conjonctives et disjonctives

On rappelle qu'une forme normale conjonctive (FNC) est une conjonction (\wedge) de disjonctions (\vee) de littéraux, chaque littéral étant une variable propositionnelle ou sa négation.

Par exemple, $(\neg a \vee b) \wedge (\neg b \vee c \vee d) \wedge c$ est une FNC.

On rappelle qu'une forme normale disjonctive (FND) est une disjonction (\vee) de conjonctions (\wedge) de littéraux, chaque littéral étant une variable propositionnelle ou sa négation.

D20. Montrer par induction structurelle sur les formules logiques que toute formule logique est équivalente à une FNC ainsi qu'à une FND.

Solution :

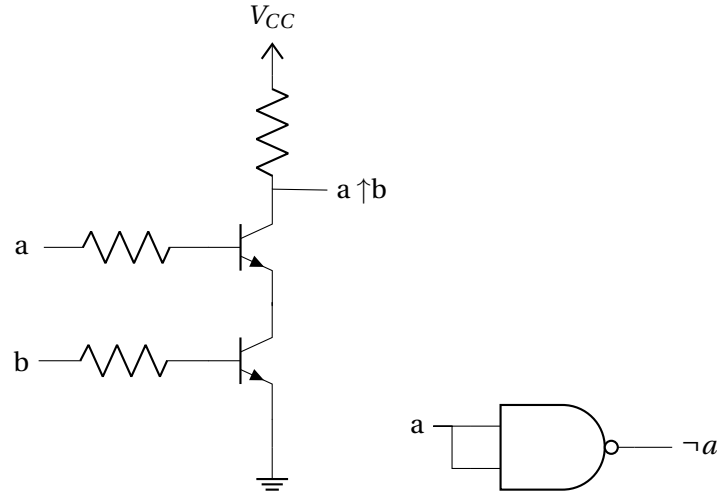


FIGURE 1 – Pour la culture électronique : le circuit d'une porte NAND à base de transistors NPN et une porte NAND schématisée et transformée en porte Non

(Cas de base) Si $\varphi = \top$ ou $\varphi = \perp$ ou $\varphi = x$, alors φ est à la fois une FND et une FNC. La propriété est donc vérifiée.

(Pas d'induction) On suppose qu'on dispose de formules ψ et ξ qui vérifient la propriété, c'est-à-dire il existe une FNC ψ' et une FND ψ'' de ψ et il existe une FNC ξ' et une FND ξ'' de ξ .

$(\neg\psi)$ Par les lois de De Morgan, $\neg\psi'$ est une FND et $\neg\psi''$ est une FNC.

$(\psi \wedge \xi)$ On remarque que $\psi' \wedge \xi'$ est toujours une FNC. Pour la FND,

$$\psi'' \wedge \xi'' = c_1 \vee c_2 \vee \dots \vee c_k \wedge d_1 \vee d_2 \vee \dots \vee d_k$$

En utilisant la distributivité de \wedge on obtient :

$$\psi'' \wedge \xi'' = (c_1 \wedge d_1) \vee (c_2 \wedge d_1) \vee \dots \vee (c_k \wedge d_1) \vee \dots = \bigvee_{i,j} (c_i \wedge d_j)$$

c'est-à-dire une forme disjonctive.

$(\psi \vee \xi)$ On remarque que $\psi'' \vee \xi''$ est toujours une FND. Pour la FNC,

$$\psi' \vee \xi' = c_1 \wedge c_2 \wedge \dots \wedge c_k \vee d_1 \wedge d_2 \wedge \dots \wedge d_k$$

. En utilisant la distributivité de \vee on obtient :

$$\psi' \wedge \xi' = (c_1 \vee d_1) \wedge (c_2 \vee d_1) \wedge \dots \wedge (c_k \vee d_1) \wedge \dots = \bigwedge_{i,j} (c_i \vee d_j)$$

c'est-à-dire une forme conjonctive.

(Conclusion) La propriété étant vérifiée dans le cas de base et invariante par les constructeurs des formules logiques, la propriété est vraie pour toute formule logique.

D21. Donner une FNC et une FND équivalente à $\neg(x \Rightarrow (\neg y \wedge z)) \vee (z \Rightarrow y)$.

Solution :

$$F = \neg(x \Rightarrow (\neg y \wedge z)) \vee (z \Rightarrow y) \equiv \neg(\neg x \vee (\neg y \wedge z)) \vee (\neg z \vee y) \equiv (x \wedge (y \vee \neg z)) \vee \neg z \vee y$$

Pour trouver une FND, on peut utiliser la distributivité de \wedge :

$$(x \wedge y) \vee (x \wedge \neg z) \vee \neg z \vee y$$

Pour trouver une FNC, on peut utiliser la distributivité de \vee :

$$F = (x \vee \neg z \vee y) \wedge (y \vee \neg z \vee \neg z \vee y) \equiv (x \vee \neg z \vee y) \wedge (y \vee \neg z)$$

On se donne des types pour représenter les formes normales **conjonctives** en OCaml :

```
1 type literal = bool * int (* si le bool est false, cela signifie la négation de la variable *)
2 type clause = literal list
3 type cnf = clause list
```

Une forme normale conjonctive est donc une liste de listes de tuples dans laquelle chaque sous-liste représente une disjonction de littéraux. Par exemple, `[(true,1);(false,3);(false,7)]` représente $x_1 \vee \neg x_3 \vee \neg x_7$. C'est pourquoi la formule f définie par :

`let f=[[(true,0);(false,2);(true,3)];[(false,0);(true,1);(true,4)];[(false,1);(true,2);(false,3)]]` représente la FNC $(x_0 \vee \neg x_2 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$.

(R) La forme normale conjonctive est une convention dans ce cas : on considère qu'on prend la conjonction de toutes les disjonctions des sous-listes. On pourrait en effet tout aussi bien représenter une forme normale disjonctive avec le même type en choisissant une autre convention.

Pour établir si une forme normale est satisfaisable, on procède par substitution par `true` ou `false` des variables. Cela permet de simplifier les clauses de la manière suivante :

- Si un littéral est `(true,i)` et que l'on substitue `true` à x_i , alors la clause disjonctive est vraie, on peut la retirer de la FNC.
- Si un littéral est `(true,i)` et que l'on substitue `false` à x_i , on peut retirer ce littéral de la clause.
- Si un littéral est `(false,i)` et que l'on substitue `false` à x_i , alors la clause disjonctive est vraie, on peut la retirer de la FNC.
- Si un littéral est `(false,i)` et que l'on substitue `true` à x_i , on peut retirer ce littéral de la clause.

On applique ensuite l'idée de base de l'algorithme de Quine : une formule logique est satisfaisable soit dans le cas où une variable est vraie, soit si cette même variable est fausse. On étudie alors les effets de la simplification des clauses disjonctives. Si, au fur et à mesure des substitutions des variables, la forme normale conjonctive est devenue :

1. une liste vide, alors la FNC est satisfaisable, car toutes les clauses sont vraies et ont été retirées.
2. une liste qui contient au moins une sous-liste vide, alors la FNC n'est pas satisfaisable car une clause au moins est fausse.

D22. Écrire une fonction de signature `nb_clauses : cnf → int` qui renvoie le nombre de clauses contenues dans une FNC.

Solution :

```
1 let rec nb_clauses phi =
2   match phi with
3   | [] → 0
4   | _ :: t → 1 + nb_clauses t;;
```

D23. Écrire une fonction récursive de signature `has_empty_clause : cnf → bool` qui statue sur le fait qu'une forme normale conjonctive contient une sous-liste vide, c'est-à-dire une clause fausse.

Solution :

```
1 let rec has_empty_clause f =
2   match f with
3   | [] → false
4   | [] :: _ → true
5   | _ :: t → has_empty_clause t;;
```

D24. Écrire une fonction récursive de signature `simplify_clause : int → bool → clause → clause option` qui simplifie une clause disjonctive d'après les remarques précisées ci-dessus : la variable (`int`) est substituée par un booléen (`bool`) et on en tire les conséquences. Cette fonction renvoie un type `option`, c'est-à-dire `None` si la clause est vraie, `Some clause` sinon. On note qu'à la fin d'une substitution, la clause peut-être vide. Par exemple, voici quelques résultats de cette fonction :

```
1 simplify_clause 0 true [(false,1);(true, 0);(false,2)];;
2 = None
3 simplify_clause 1 false [(false,1);(true, 0);(false,2)];;
4 = None
5 simplify_clause 1 true [(false,1);(true, 0);(false,2)];;
6 = Some [(true, 0); (false, 2)]
7 simplify_clause 0 false [(true, 0);(false,2)];;
8 = Some [(false, 2)]
9 simplify_clause 2 true [(false,2)];;
10 = Some []
```

D25. Écrire une fonction récursive de signature `simplify_cnf : int → bool → cnf → cnf` qui simplifie une forme normale conjonctive en simplifiant les clauses disjonctives qui la composent. Par exemple, pour la formule f définie ci-dessus on a :

```
1 simplify_cnf 2 true f;;
2 [[(true, 0); (true, 3)]; [(false, 0); (true, 1); (true, 4)]]
3 simplify_cnf 2 false f;;
4 [[(false, 0); (true, 1); (true, 4)]; [(false, 1); (false, 3)]]
```

Solution :

```

1 let rec simplify_cnf k v f =
2   match f with
3   | [] -> []
4   | cl :: t -> let c = simplify_clause k v cl in
5                 match c with
6                 | None -> simplify_cnf k v t
7                 | Some lst -> lst :: (simplify_cnf k v t);;

```

D26. Écrire une fonction récursive de signature `sat_cnf : cnf -> bool` qui statue sur la satisfaisabilité d'une forme normale conjonctive en appliquant les remarques précisées ci-dessus.

Solution :

```

1 let rec sat_cnf f =
2   if has_empty_clause f
3   then false
4   else match f with
5   | [] -> true
6   | cl :: _ -> let (_,k) = List.hd cl in
7                 sat_cnf (simplify_cnf k true f) || sat_cnf (simplify_cnf k false f);;

```

D27. Quel est le résultat de `sat_cnf` appliqué à la formule `let phi = [[(true, 0)];[(false,1);(false,2)];[(false,0);(true,2)];[(true,1);(false,0)]]`?

Solution : Cette formule est une antilogie, donc pas satisfaisable. Le résultat sera donc `false`.