# Dans les pas d'Indy

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 déchiffrage 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 $A_3$.

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.

**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 posez les pieds pour ouvrir cette porte.

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.

**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.

## B Tautologie, équivalence et évaluation

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

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

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

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

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

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

On dispose du type amgébrique récursif OCaml suivant :

```
type formule =
  | T (* true *)
  | F (* false *)
  | Var of int (* variable *)
  | Not of formule (* negation *)
  | And of formule * formule (* conjonction *)
  | Or  of formule * formule  (* disjonction *)
  | Imp of formule * formule (* implication *)
  | 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 $\phi = (a \wedge b) \Rightarrow c \iff a \Rightarrow (b \Rightarrow c)$.

**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.

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 $1000_2 = 2^3 = 8$.

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

**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`.

## C Système complet logique

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

| a | b | a ↑ b | a ↓ b | a ⊕ 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 $\neg$.

**C14**. Montrer que :

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

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

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

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

**C16**. Montre que :

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

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

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

(d) Que peut-on en déduire?

**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 structurelle sur les formules logiques que pour une formule $\varphi$ exprimée uniquement avec des $\oplus$ et des variables propositionnelles, $[\![\varphi]\!]_d = 0$.

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

On dispose des types OCaml suivants :

```
type formule =
  | T (* true *)
  | F (* false *)
  | Var of int (* variable *)
  | Not of formule (* negation *)
  | And of formule * formule (* conjonction *)
  | Or  of formule * formule  (* disjonction *)

type nand_formule =
    | NT (* true *)
    | NF (*false*)
    | NVar of int (* x_k *)
    | 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 $\uparrow$.

**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 $\{\neg, \wedge, \vee\}$.

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.

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

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

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

---

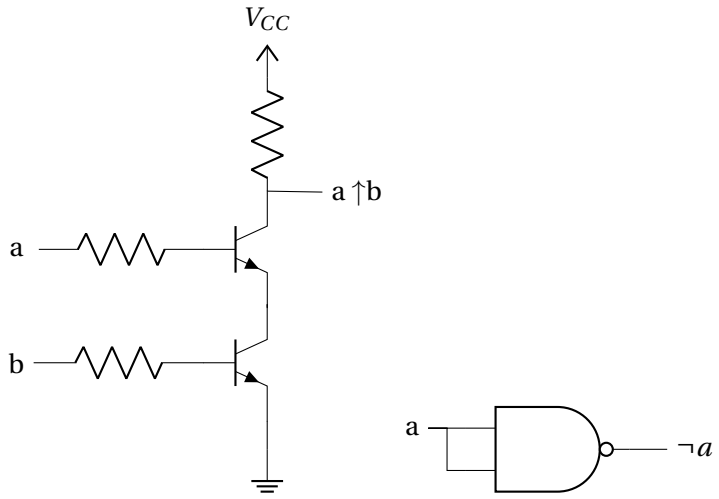


FIGURE 1 – Pour la culture électronicienne : le circuit d'une porte NAND à base de transistors NPN et une porte NAND schématisée et transformée en porte Non

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.

**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.

**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 :

```
simplify_clause 0 true [(false,1);(true, 0);(false,2)];;
= None
simplify_clause 1 false [(false,1);(true, 0);(false,2)];;
= None
simplify_clause 1 true [(false,1);(true, 0);(false,2)];;
= Some [(true, 0); (false, 2)]
simplify_clause 0 false [(true, 0);(false,2)];;
= Some [(false, 2)]
simplify_clause 2 true [(false,2)];;
= 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 :

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

---

**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.

**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)]]`?