Rappels de logique

OPTION INFORMATIQUE - Devoir n° 3.3m - Olivier Reynet

A Rappels

■ **Définition 1 — Algèbre de Boole .** L'algèbre de Boole est un ensemble à deux éléments que l'on peut noter de différentes manières :

$$\mathbb{B} = \{0, 1\} = \{\text{Faux}, \text{Vrai}\} = \{\bot, \top\} \tag{1}$$

Cet ensemble est muni des opérateurs suivants :

1. la négation notée $\neg$ (opérateur unaire)
2. la conjonction notée $\wedge$
3. la disjonction notée $\vee$
4. la disjonction exclusive notée $\veebar$
5. l'implication notée $\Longrightarrow$
6. l'équivalence notée $\Longleftrightarrow$

Vocabulary 1 — Bottom $\bot$ et Top $\top$ $\leftrightsquigarrow$ Faux et Vrai

On considère un ensemble de variables propositionnelles $\mathcal{V}$ utilisé pour écrire un ensemble de formules $\mathcal{F}$ en logique propositionnelle.

■ **Définition 2 — Ensemble des formules propositionnelles (défini inductivement).** L'ensemble $\mathcal{F}$ des formules propositionnelles sur $\mathcal{V}$ est défini inductivement comme suit :

$$\forall e \in \mathbb{B} = \{\bot, \top\}, e \in \mathcal{F} \tag{2}$$
$$\forall v \in \mathcal{V}, v \in \mathcal{F} \tag{3}$$
$$\phi \in \mathcal{F} \Longrightarrow \neg\phi \in \mathcal{F} \tag{4}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, \forall \diamond \in \{\wedge, \vee, \Longrightarrow, \Longleftrightarrow\}, \phi \diamond \psi \in \mathcal{F} \tag{5}$$

R Cette définition inductive permet de représenter une formule logique par un arbre binaire, l'arbre syntaxique. Elle permet également de démontrer de nombreux résultats.

■ **Définition 3 — Valuation ou interprétation.** Une valuation de $\mathcal{V}$ est une distribution des valeurs de vérité sur l'ensemble des variables propositionnelles $\mathcal{V}$, soit une fonction $v : \mathcal{V} \longrightarrow \mathbb{B}$.

■ **Définition 4 — Évaluation d'une formule logique (définie inductivement).** Soit v une valuation de $\mathcal{V}$. L'évaluation d'une formule logique d'après v est notée $[\![\phi]\!]_v$. Elle est définie inductivement par :

$$[\![\bot]\!]_v = \bot \tag{6}$$
$$[\![\top]\!]_v = \top \tag{7}$$
$$\forall x \in \mathcal{V}, [\![x]\!]_v = v(x) \tag{8}$$
$$\forall \phi \in \mathcal{F}, [\![\neg\phi]\!]_v = \neg[\![\phi]\!]_v \tag{9}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, [\![\phi \wedge \psi]\!]_v = [\![\phi]\!]_v \wedge [\![\psi]\!]_v \tag{10}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, [\![\phi \vee \psi]\!]_v = [\![\phi]\!]_v \vee [\![\psi]\!]_v \tag{11}$$
$$\tag{12}$$

■ **Définition 5 — Déduction $\vdash$.** Si on peut déduire logiquement ϕ de ψ, alors on écrit $\psi \vdash \phi$.

■ **Définition 6 — Modèle.** Un modèle pour $\mathcal{F}$ est une valuation v telle que :

$$\forall \phi \in \mathcal{F}, [\![\phi]\!]_v = \top \tag{13}$$

■ **Définition 7 — Conséquence sémantique $\models$.** Soit ϕ une formule de $\mathcal{F}$. On dit que ϕ est une conséquence de $\mathcal{F}$ si tout modèle de $\mathcal{F}$ est un modèle de ϕ. On note alors : $\mathcal{F} \models \phi$

■ **Définition 8 — Tautologie.** Une formule ϕ toujours évaluée à $\top$ quel que soit le modèle d'interprétation est une tautologie. On la note $\top$ ou $\models \phi$.

■ **Définition 9 — Antilogie.** Une formule ϕ toujours évaluée à $\bot$ quel que soit le modèle d'interprétation est une antilogie. On la note $\bot$.

■ **Définition 10 — Formule satisfaisable.** Soit ϕ une formule logique de $\mathcal{F}$. S'il existe une valuation v de $\mathcal{V}$ qui satisfait ϕ, alors ϕ est dite satisfaisable.

Plus formellement :

$$\phi \text{ est une formule satisfaisable} \iff \exists v : \mathcal{V} \longrightarrow \mathbb{B}, [\![\phi]\!]_v = \top \tag{14}$$

■ **Définition 11 — Équivalence de formules.** Deux formules logiques sont équivalentes si, quelle que soit la valuation choisie, l'évaluation des formules sont égales.

Plus formellement, si ϕ et ψ sont des formules logiques de $\mathcal{F}$, on a :

$$\phi \equiv \psi \iff \forall v : \mathcal{V} \longrightarrow \mathbb{B}, [\![\phi]\!]_v = [\![\psi]\!]_v \tag{15}$$

Théorème 1 — Lois de Morgan.

$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, \neg(\phi \wedge \psi) \equiv \neg\psi \vee \neg\psi \tag{16}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, \neg(\phi \vee \psi) \equiv \neg\psi \wedge \neg\psi \tag{17}$$

■ **Définition 12 — Littéral.** Un littéral est une variable ou sa négation.

■ **Définition 13 — Clause.** Une clause est une conjonction de littéraux.

B Des sphinx, des règles et des énigmes

Thésée a pris le chemin d'Athène. Sur la route, des sphinx lui barrent le passage. La seule manière de passer est de répondre à leurs énigmes logiques.

L'énonciation des énigmes par les sphinx suit une règle qui est systématiquement respectée. Lorsque les sphinx énoncent une règle, ils disent toujours la vérité et voici ce que le premier Sphinx a dit à Thésée :

> «Lorsque nous énonçons les énigmes, nous pouvons soit dire la vérité, soit mentir. Mais, pour une énigme donnée, la première et la dernière de nos affirmations sont de la même nature, soit vérité, soit mensonge. Toutes les autres affirmations sont de la nature opposée à ces deux-là.»

B1. Un des sphinx fait une suite de n déclarations D_i dans une même énigme. Proposer une formule du calcul des propositions qui représente la règle qu'il respecte.

Solution : $D = D_1 \wedge \neg D_2 \wedge \ldots \wedge \neg D_{n-1} \wedge D_n \vee \neg D_1 \wedge D_2 \wedge \ldots \wedge D_{n-1} \wedge \neg D_n$

C Énigme à deux sphinx

Thésée se retrouve face à deux sphinx et à trois escaliers, l'un à gauche, l'autre à droite et le dernier au milieu entre les deux autres.

Le premier Sphinx P énonce les affirmations suivantes :

1. l'escalier de gauche est sûr,
2. l'escalier du milieu est sûr ou celui de droite n'est pas sûr.

Le second Sphinx S énonce les affirmations suivantes :

1. ni l'escalier de gauche, ni celui du milieu ne sont sûrs,
2. si les escaliers de gauche ou de droite sont sûrs, alors l'escalier du milieu est sûr.

On note G, M et D les variables propositionnelles associées au fait que les escaliers de gauche, du milieu et de droite sont sûrs. P1 et P2, respectivement S1 et S2, sont les formules propositionnelles associées aux déclarations du premier sphinx, respectivement du second sphinx.

On note P, respectivement S, la formule du calcul des propositions dépendant des variables P1 et P2, respectivement S1 et S2, qui correspond au respect de la règle par le premier sphinx, respectivement par le second sphinx, dans cette énigme.

On note R la formule du calcul des propositions dépendant des variables P et S qui décrit le respect global des règles par les deux Sphinx dans cette énigme.

C1. Représenter les déclarations des deux Sphinx sous la forme de formules du calcul des propositions P1, P2, S1 et S2 dépendant des variables G, M et D

Solution :

$$P1 = G \tag{18}$$
$$P2 = M \vee \neg D \tag{19}$$
$$S1 = \neg G \wedge \neg M \tag{20}$$
$$S2 = (G \vee D \Longrightarrow M) = (\neg G \wedge \neg D) \vee M \tag{21}$$

C2. Appliquer la règle respectée par les sphinx que vous avez proposée pour la première question (trouver P, S et R).

Solution :

$$P = (P1 \wedge P2) \vee (P1 \wedge P2) \tag{22}$$
$$S = (S1 \wedge S2) \vee (S1 \wedge S2) \tag{23}$$
$$R = P \wedge S \tag{24}$$

C3. En utilisant une table de vérité, déterminer quel est (ou quels sont) le (ou les) escalier(s) qui est (ou sont) sûr(s), c'est à dire à quelles conditions R est vraie ? Vous indiquerez explicitement les résultats intermédiaires correspondant aux formules P1, P2, P, S1, S2 et S.

Solution : L'escalier de gauche est le seul escalier que doit emprunter Thésée.

D Énigme à un seul sphinx

Thésée rencontre ensuite un autre sphinx devant trois portes. Celui-ci affirme :

1. la porte rouge n'est pas sûre ou la porte verte est sûre,
2. si les portes rouge et verte sont sûres, alors la porte bleue n'est pas sûre,
3. la porte verte n'est pas sûre mais la porte bleue est sûre.

On note P3, P4 et P5 les formules propositionnelles associées aux déclarations du sphinx et R, V et B les variables propositionnelles associées au fait que les portes rouge, verte et bleue sont sûres.

On note P la formule du calcul des propositions dépendant des variables P3, P4 et P5 qui correspond au respect de la règle d'énonciation de cette énigme.

D1. Représenter les déclarations du Sphinx sous la forme de formules du calcul des propositions P3, P4 et P5 dépendant des variables R, V et B.

Solution :

$$P3 = \neg R \vee V \tag{25}$$
$$P4 = (R \wedge V \Longrightarrow \neg B) = \neg R \vee \neg V \vee \neg B \tag{26}$$
$$P5 = \neg V \wedge B \tag{27}$$

D2. Appliquer la règle d'énonciation des énigmes (trouver P).

Solution :

$$P = P3 \wedge \neg P4 \wedge P5 \vee \neg P3 \wedge P4 \wedge \neg P5 \tag{28}$$

D3. En utilisant le calcul des propositions, déterminer quelle est (ou quelles sont) la (ou les) porte(s) qui est (ou sont) sûre(s).

Solution :

$$\begin{aligned} P &= (\neg R \vee V) \wedge \neg(\neg R \vee \neg V \vee \neg B) \wedge \neg V \wedge B \vee (\neg\neg R \vee V) \wedge (\neg R \vee \neg V \vee \neg B)\neg\neg V \wedge B && (29)\\ &= (\neg R \vee V)R \wedge V \wedge B \wedge \neg V \wedge B \vee R \wedge \neg V \wedge (\neg R \vee \neg V \vee \neg B) \wedge (V \vee \neg B) && (30)\\ &= R \wedge \neg V \wedge \neg B \wedge (\neg R \vee \neg V \vee \neg B) && (31)\\ &= R \wedge \neg V \wedge \neg B && (32) \end{aligned}$$

E Évaluation d'une formule logique

On dispose d'un type `formule` qui permet d'écrire des formule en logique propositionnelle en OCaml.

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

E1. Utiliser le type `formule` pour encoder la formule logique $\neg a \vee (b \wedge c)$. On proposera deux versions `fc` et `fi` selon le type choisi pour `'a` : on pourra choisir de modéliser les variables propositionnelles par des `char` (`'a', 'b','c'`) ou par des `int` (`0,1,2`).

Solution :

```
let fc = Or ( (Var 'a') ,(And  ((Var 'b') , (Var 'c'))) );;
let fi = Or ( (Var 0) ,(And  ((Var 1) , (Var 2))) );;
```

E2. Écrire une fonction de valuation de signature `char_valuation : char -> bool` dont le paramètre est une variable sous la forme d'un `char` et qui renvoie un booléen ou échoue avec le message `" Unknown variable !"` si la variable n'existe pas. Choisir arbitrairement les valeurs affectées aux variables `'a', 'b','c'`.

Solution :

```
let char_valuation c = match c with
    | 'a' -> true
    | 'b' -> false
    | 'c' -> true
    |  _ -> failwith "Unknown variable !";;
```

E3. Écrire une fonction de valuation de signature `int_valuation : int -> bool` dont le paramètre est une variable sous la forme d'un `int` et qui renvoie un booléen ou échoue avec le message `"Unknown variable !"` si la variable n'existe pas. Choisir arbitrairement les valeurs affectées aux variables `0, 1, 2`.

Solution :

```
let int_valuation i = match i with
  | 0 -> true
  | 1 -> false
  | 2 -> true
  | _ -> failwith "Unknown variable !";;
```

E4. En utilisant la définition inductive de l'évaluation, programmer une fonction de signature `evaluation : ('a -> bool)-> 'a formule -> bool` dont les paramètres sont une valuation et une formule et qui renvoie un booléen. Tester cette fonction sur les formules `fc` et `fi` et les valuations `char_valuation` et `int_valuation`.

Solution :

```
let rec evaluation v f =
  match f with
    | T -> true
    | F -> false
    | Var x -> v x
    | Not p -> not (evaluation v p)
    | And (p, q) -> evaluation v p && evaluation v q
    | Or (p, q) -> evaluation v p || evaluation v q
;;
```

E5. À l'aide de la fonction précédente, évaluer la proposition R de la question A2 avec la valuation qui donne la solution de l'énigme.

Solution :

```
let escalier_valuation x =
  match x with
    | 'G' -> true
    | 'M' -> false
    | 'D' -> false
    | _ -> failwith "Unknown variable !";;
```

```
7
8   let r =
9       let p1 = Var 'G'
10        and p2 = Or (Var 'M', Not(Var 'D'))
11        and s1 = And(Not(Var 'G'), Not(Var 'M'))
12        and s2 = Or (Var 'M', And( Not (Var 'G'), Not (Var 'D')))
13      in let p = Or ( And (p1, p2), And(Not p1, Not p2) )
14        and s = Or (And(s1, s2), And(Not s1, Not s2) )
15    in And (p, s);;
16
17      evaluation escalier_valuation r;;
```

F Satisfaisabilité par la force brute

On choisit de représenter une valuation par un nombre entier. Chaque bit de ce nombre entier représente une variable propositionnelle de l'ensemble $\mathcal{V}$. On choisit de représenter $\bot$ par un bit à 0 et $\top$ par un bit à 1.

F1. Écrire une fonction de signature `var_k_valuation : int -> int -> bool` dont les paramètres sont :

- un type `int` j représentant une valuation (de toutes les variables),
- un type `int` k représentant l'indice d'une variable propositionnelle valuée.

Cette fonction renvoie un booléen qui statue sur la valeur (0 ou 1) de la variable propositionnelle d'indice k. Il est nécessaire d'utiliser des fonctions de la bibliothèque `Int` :

- `val logand : int -> int -> int` : logand x y is the bitwise logical and of x and y.
- `val shift_left : int -> int -> int` : `shift_left x n` shifts x to the left by n bits.

R En OCaml, pour calculer une puissance de 2, on utilise volontiers `Int.shift_left` car l'opérateur `**` ne s'applique qu'à des types `float`.

Solution :

```
1     let var_k_valuation j k =  Int.logand j (Int.shift_left 1 k) != 0 ;;
```

On veut tester la satisfaisabilité d'une formule logique par la force brute, c'est à dire qu'on évalue la formule pour toutes les valuations possibles. Dès qu'on trouve une valuation qui la satisfait, alors on renvoie cette valuation.

F2. Écrire une fonction de signature `brute_force_satisfiability : int formule -> int -> int option` dont les paramètres sont :

- une formule logique de type `int formule`,
- un type `int` représentant le numéro/indice de la variable de la formule.

Cette fonction renvoie un type option `None` si la formule n'est pas satisfaisable ou l'entier correspondant à la première valuation trouvée pour laquelle la formule est satisfaisable.

Solution :

```
let brute_force_satisfiability f n = (* n : nombre de variables de type int
    de 0 à n-1 *)
  let v_limit = Int.shift_left 1 n in
  let rec check_val v =
      match v with
        | k when k < v_limit -> if evaluation (var_k_valuation v) f
                                then Some v
                                else check_val (v + 1);
        | _ -> None
  in check_val 0;;
```

F3. Quelle est la complexité de la fonction `brute_force_satisfiability` ?

F4. Résoudre l'énigme des deux premiers sphinx (question A3).

Solution :

```
let ri =
  let p1 = Var 0
    and p2 = Or (Var 1, Not(Var 2))
    and s1 = And(Not(Var 0), Not(Var 1))
    and s2 = Or (Var 1, And( Not (Var 0), Not (Var 2)))
  in let p = Or ( And (p1, p2), And(Not p1, Not p2) )
    and s = Or (And(s1, s2), And(Not s1, Not s2) )
  in And (p, s);;
brute_force_satisfiability ri 3;;
```

G Formes normales

G1. Montrer que toute formule logique est équivalente à une forme normale disjonctive (FND).

Solution : En fait, il suffit d'écrire que cette formule logique est la disjonction de toutes ses valuations vraies. Plus formellement :

$$\bigvee_{\substack{v,\\ [\![\phi]\!]_v=\top}} \bigwedge_{\substack{x\in\mathcal{V},\\ v(x)=\top}} x \tag{33}$$

G2. Montrer que toute formule logique est équivalente à une forme normale conjonctive (FNC). On pourra utiliser la question précédente et les lois de Morgan.

Solution : Soit ϕ une formule logique. On considère sa négation $\neg\phi$. D'après la question précédente, on peut mettre $\neg\phi$ sous une forme normale disjonctive, c'est à dire

$$\neg\phi = c_1 \vee c_2 \ldots \vee c_n \tag{34}$$

où les $c_i = l_1 \wedge l_2 \wedge \ldots \wedge l_m$ sont des conjonctions de littéraux. En appliquant la loi de Morgan, on trouve que :

$$\begin{aligned} \neg\neg\phi &= (\neg c_1) \wedge (\neg c_2) \wedge \ldots \wedge (\neg c_n) && (35)\\ &= (l_1 \vee l_2 \ldots \vee l_m) \wedge (\neg c_2) \wedge \ldots \wedge (\neg c_n) && (36)\\ &= d_1 \wedge d_2 \wedge \ldots \wedge d_n && (37)\\ &= \phi && (38) \end{aligned}$$

où les d_i sont des disjonctions. Donc ϕ peut s'écrire sous une forme normale conjonctive.

G3. Donner une FNC équivalente à $(a \vee \neg b) \wedge \neg(c \wedge \neg(d \wedge e))$.

Solution : $(a \vee \neg b) \wedge (\neg c \vee d) \wedge (\neg c \vee e)$

H Algorithme de Quine

L'algorithme de Quine, c'est le retour sur trace appliqué à au problème SAT. L'algorithme de Quine suppose que la formule logique est donnée sous une forme normale conjonctive et manipule donc la liste des clauses de cette forme. Il est nécessaire, pour que la formule soit satisfaisable, que toutes les clauses soient vraies.

Les solutions partielles (le nœuds de l'arbre de recherche) sont créées en substituant à chaque variable $\bot$ ou $\top$ dans chaque clause. Une fois substituée, il est possible de simplifier une solution partielle comme suit :

Si on a substitué $\top$ à la variable a alors :

- Si une clause contient a, on retire cette clause car elle est vraie, car $\top$ est l'élément absorbant de la disjonction,
- Si une clause contient $\neg a$, on retire $\neg a$ de cette clause car $\bot$ est l'élément neutre d'une disjonction.

L'algorithme fonctionne comme suit :

- Si une clause est vide après substitution, alors la formule n'est pas satisfaisable en l'état car toutes les variables sont fausses. On effectue le retour sur trace.
- Si l'ensemble des clauses est vide, alors la formule elle est vraie : on a trouvé une solution.

On se donne les éléments de code OCaml suivants :

```
type 'a litteral =
| Var of 'a (* variable *)
| Neg of 'a (* negation *);;

let get_var a = match a with (* récupérer le nom/numéro de la variable *)
  | (Var v) -> v
  | (Neg v) -> v;;

type 'a clause = 'a litteral list;;
type 'a cnf = 'a clause list;;

exception True_clause_detected;;
```

Algorithme 1 Algorithme Quine

```
1 : Fonction QUINE(𝒞)                                   ▷ 𝒞 est l'ensemble de clauses de la FNC
2 :     si 𝒞 est vide alors
3 :         renvoyer Vrai                  ▷ On a réussi à retirer toutes les clauses car elles étaient vraies
4 :     si une des clauses de 𝒞 est vide alors
5 :         renvoyer Faux                                ▷ Toutes les variables étaient fausses
6 :     Choisir une variable propositionnelle a dans une clause
7 :     si QUINE(𝒞[a ← ⊤]) alors                        ▷ On substitue ⊤ à a dans les clauses
8 :         renvoyer Vrai
9 :     sinon
10 :        renvoyer QUINE(𝒞[a ← ⊥])                    ▷ Si solution, alors a vaut ⊥
```

H1. Écrire une fonction récursive de signature `sub_clause : 'a litteral list -> 'a litteral -> bool -> 'a litteral list` qui transforme une clause conformément à l'algorithme de Quine. Les paramètres sont :

- une clause à transformer,
- un littéral,
- un booléen.

`sub_clause` substitue dans la clause le littéral par sa valeur booléenne.

- Lorsqu'une clause est vide, la fonction renvoie une liste vide.
- Lorsqu'une clause est vraie, elle sera retirée de la forme normale. Dans ce but, la fonction lève une exception `True_clause_detected`.
- Dans tous les autres cas, la fonction renvoie la clause modifiée.

Solution :

```
let rec sub_clause c a value =
  let av = get_var a in
    match c with
    | [] -> []
    | (Var x)::_ when x = av && value -> raise True_clause_detected
    | (Neg x)::t when x = av && value -> sub_clause t a value
    | (Var x)::t when x = av && not value -> sub_clause t a value
    | (Neg x)::_ when x = av && not value -> raise True_clause_detected
    | x::t -> x::(sub_clause t a value);;
```

H2. Écrire une fonction récursive de signature `sub_cnf : 'a litteral list list -> 'a litteral -> bool -> 'a litteral list list` qui effectue les substitutions nécessaires sur toutes les clauses d'une forme normale conjonctive. Cette fonction utilise la fonction précédente. Si une exception `True_clause_detected` est levée, `sub_cnf` continue de traiter le reste de la forme normale : la clause vraie n'apparaît donc plus dans la forme normale. Pour cela, il est nécessaire d'utiliser `try .. with`.

Solution :

```
let rec sub_cnf cnf a value =
  match cnf with
    | [] -> []
    | c::t -> try (sub_clause c a value)::(sub_cnf t a value) with
            | True_clause_detected -> sub_cnf t a value;;

sub_cnf [[Var 0; Neg 1]; [Neg 0; Var 1]; [ Neg 0; Neg 1]; [Var 0; Neg 2]] (
    Var 0) true;;
```

H3. Écrire une fonction récursive de signature `has_empty_list : 'a list list -> bool` qui teste si une liste de liste possède des listes vides. Elle renvoie `true` si une liste vide est présente dans la liste de liste.

Solution :

```
let rec has_empty_list l =
  match l with
    | [] -> false
    | h::_ when h = [] -> true
    | _::t -> has_empty_list t;;
```

H4. Écrire une fonction récursive de signature `quine : 'a litteral list list -> bool` qui implémente l'algorithme de Quine en se servant des fonctions précédentes. Le choix d'une variable à substituer dans une clause peut se faire simplement en prenant la première variable [1].

Solution :

```
let quine cnf =
  let rec aux v =
    match v with
      | [] -> true
      | forme when has_empty_list forme -> false (* Backtracking ! *)
      | c::_ -> let p = List.hd c in  (* on peut faire un meilleur choix *)
           if aux (sub_cnf v p true) then true else aux (sub_cnf v p false)
  in aux cnf;;

quine [[Var 0]; [Var 1]];;
quine [[Var 0; Neg 1]; [Neg 0; Var 1]; [ Neg 0; Neg 1]; [Var 0; Neg 2]; [Var
     0; Var 1]];;
```

H5. Dans le pire des cas, quelle est la complexité de cet algorithme ?

Solution : Cette complexité est exponentielle, on parcourt l'arbre de recherche en entier.

1. L'algorithme est cependant plus performant si on choisit la variable la plus présente dans les clauses.

H6. On peut améliorer l'algorithme de Quine en observant que si la clause ne contient plus qu'un seul littéral, alors on peut propager la valeur de ce littéral dans les autres clauses. Implémenter cette amélioration !

Solution :

```
let dpll cnf =
  let rec aux v =
  match v with
    | [] -> true
    | forme when has_empty_list forme -> false (* Backtracking ! *)
    | [Var b]::_ ->  aux (sub_cnf v (Var b) true)
    | [Neg b]::_ ->  aux (sub_cnf v (Var b) false)
    | c::_ -> let p = List.hd c in  (* on peut faire un meilleur choix *)
          if aux (sub_cnf v p true) then true else aux (sub_cnf v p false)
in aux cnf;;
```