Chapitre 5

option informatique

Logique des propositions

1. Introduction

D'un point de vue formel, une logique est définie par une *syntaxe*, c'est à dire la donnée d'un ensemble de symboles et de règles. Il s'agit donc d'un langage (dont les mots sont appelés les *formules logiques*), à qui on associe une *sémantique* permettant d'attribuer une valeur (le vrai ou le faux) aux symboles et aux formules.

On distingue deux types de logique : la logique des propositions, qui définit les lois formelles du raisonnement, et la logique des prédicats, qui formalise le langage des mathématiques en s'autorisant l'usage de quantificateurs (en général $\forall$ et $\exists$). Par exemple, « (a ou b) $\Rightarrow b$ » est une formule de la logique propositionnelle, alors que « $\exists x \mid (x$ ou $b) \Rightarrow b$ » relève de la logique des prédicats.
La sémantique permet d'attribuer aux variables libres une valeur (vrai ou faux) et d'en déduire à l'aide de règles de calcul la valeur d'une formule. Par exemple, pour a = vrai et b = faux la formule « (a ou b) $\Rightarrow b$ » est fausse. En revanche, pour b = faux la formule « $\exists x \mid (x$ ou $b) \Rightarrow b$ » est vraie [1].

Dans la suite de ce cours, nous nous intéresserons exclusivement à la logique propositionnelle.

2. Logique des propositions

2.1 Formules logiques

Pour définir le langage de la logique des propositions, on utilise un alphabet Σ constitué :

- de *constantes* Faux et Vrai qu'on notera 0 et 1 ;
- de *variables* en nombre dénombrable, représentées par les lettres romaines a, b, c ... ;
- d'un connecteur unaire $\neg$ (appelé le connecteur logique de *négation*) ;
- de quatre connecteurs binaires $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ appelés respectivement les connecteurs logiques de *conjonction* (et), *disjonction* (ou), *implication* et *équivalence*.

Une formule logique se définit alors à l'aide des règles suivantes :

- toute constante et toute variable est une formule ;
- si F est une formule, $(\neg F)$ est une formule ;
- si F_1 et F_2 sont des formules et $\triangle$ un opérateur binaire, alors $(F_1 \triangle F_2)$ est une formule.

Remarque. L'expression infixe des formules logiques nécessite bien évidemment l'usage de parenthèses pour lever toute ambiguïté sémantique, mais les règles de priorité usuelles entre opérateurs permettent d'éviter d'en écrire un trop grand nombre. Il aurait aussi été possible d'utiliser une écriture postfixée pour en éviter l'usage, mais c'est assez peu conforme aux usages.

Dans la suite de ce cours, on conviendra que l'opérateur de négation $\neg$ a priorité sur les autres, que la conjonction et la disjonction ont priorité sur l'implication et l'équivalence, et enfin que la conjonction a priorité sur la disjonction.
Par exemple, $\Big(\big((\neg a) \wedge b\big) \vee c\Big) \Rightarrow (a \wedge c)$ s'écrira plus simplement $\neg a \wedge b \vee c \Rightarrow a \wedge c$, et $\big((\neg a) \Rightarrow b\big) \vee \big(a \Leftrightarrow (b \wedge c)\big)$ se simplifiera en $(\neg a \Rightarrow b) \vee (a \Leftrightarrow b \wedge c)$.

Enfin, notons que de cette définition découle naturellement une représentation arborescente d'une formule logique. Par exemple, la dernière formule est représentée par l'arbre de la figure 1.

1. pour la logique des prédicats, les variables associées aux quantificateurs, dans notre exemple x, sont dites *liées* ; il n'en n'existe pas en logique propositionnelle, où toutes les variables sont libres.

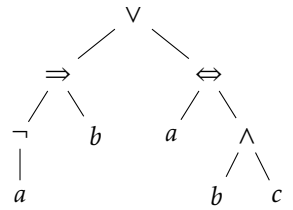


FIGURE 1 – Une représentation arborescente de la formule $(\neg a \Rightarrow b) \vee (a \Leftrightarrow b \wedge c)$.

2.2 Sémantique

On appelle *contexte* d'une formule logique une application qui, à chacune des variables présentes dans cette formule, associe la valeur 0 ou 1 (pour le Faux et le Vrai).

À tout contexte correspond une *évaluation* de la formule logique, définie inductivement à l'aide des *tables de vérité* associées à chacun des opérateurs logiques :

a	$\neg a$
0	1
1	0

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

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

a	b	$a \Leftrightarrow b$
0	0	1
0	1	0
1	0	0
1	1	1

Plus généralement, on peut associer à toute formule logique à n variables une table de vérité ayant 2^n lignes et reportant pour chaque contexte l'évaluation de la formule correspondante. Par exemple, l'assertion composée $\neg b \Rightarrow a \wedge \neg b$ est associée à la table de vérité suivante (dans laquelle on a fait apparaître l'évaluation des formules intermédiaires) :

a	b	$\neg b$	$a \wedge \neg b$	$\neg b \Rightarrow a \wedge \neg b$
0	0	1	0	0
0	1	0	0	1
1	0	1	1	1
1	1	0	0	1

Par l'intermédiaire de leurs tables de vérité, il est possible de définir 16 connecteurs binaires différents. Aux quatre opérateurs déjà définis on rajoutera le XOR (ou exclusif), le NAND (non et) et le NOR (non ou), peu usités en mathématique mais beaucoup plus en informatique, et définis par l'intermédiaire des tables de vérité suivantes :

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

• Formules logiques équivalentes

Deux formules logiques F_1 et F_2 sont dites *logiquement équivalentes* lorsque leurs tables de vérité coïncident, c'est à dire lorsque leurs évaluations coïncident dans tout contexte. On note alors : $F_1 \equiv F_2$. Par exemple, la table de vérité obtenue plus haut montre que la formule $\neg b \Rightarrow a \wedge \neg b$ est logiquement équivalente à la formule $a \vee b$.

Nous utilisons fréquemment certaines équivalences logiques dans nos raisonnements mathématiques. On peut citer par exemple :

- $(a \Rightarrow b) \equiv (\neg b \Rightarrow \neg a)$ (le raisonnement par contraposée) ;
- $a \equiv (\neg a \Rightarrow 0)$ (le raisonnement par l'absurde) ;
- $(a \Leftrightarrow b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$;
- $(a \Rightarrow b \vee c) \equiv (a \wedge \neg b \Rightarrow c)$.

Notons pour finir qu'en procédant par induction structurelle, on peut démontrer le résultat suivant :

Théorème (principe de substitution). — *Si $a_1, a_2, \dots, a_n$ sont des variables et $\mathrm{F}_1(a_1, \dots, a_n)$ et $\mathrm{F}_2(a_1, \dots, a_n)$ deux formules logiquement équivalentes faisant intervenir ces variables, alors quelles que soient les formules logiques $f_1, \dots, f_n$, les formules $\mathrm{F}_1(f_1, \dots, f_n)$ et $\mathrm{F}_2(f_1, \dots, f_n)$ restent logiquement équivalentes.*

• Satisfiabilité et tautologies

Les formules logiques qui prennent la valeur vrai dans tout contexte sont des *tautologies*. Par exemple : $a \vee \neg a$ (le *tiers exclu*), $\neg(a \wedge \neg a)$ (la *non contradiction*), $\Big((a \Rightarrow b) \wedge (b \Rightarrow c)\Big) \Rightarrow (a \Rightarrow c)$ (la transitivité de l'implication) sont des tautologies.

Remarque. Deux formules logiques F_1 et F_2 sont équivalentes si et seulement si $(\mathrm{F}_1 \Leftrightarrow \mathrm{F}_2)$ est une tautologie.

Une formule logique dont la négation n'est pas une tautologie est dite *satisfiable*. Autrement dit, il existe au moins un contexte pour lequel la formule logique prend la valeur vrai.
Pour reconnaître une tautologie ou vérifier la satisfiabilité d'une formule logique, il suffit de dresser sa table de vérité. Malheureusement, celle-ci a une croissance exponentielle en fonction du nombre n de variables qui compose la formule puisqu'elle comporte 2^n lignes. L'existence (ou la preuve de la non-existence) d'un algorithme de coût polynomial est un des grands problèmes ouverts de l'informatique d'aujourd'hui. Nous reviendrons sur cette problématique à la fin de ce chapitre.

• Lois de De Morgan

La construction des tables de vérités permet de prouver sans peine les deux théorèmes suivants :

Théorème (De Morgan). — *Si a et b sont deux variables propositionnelles, on dispose des équivalences suivantes :*

$$\neg(a \wedge b) \equiv \neg a \vee \neg b \qquad et \qquad \neg(a \vee b) \equiv \neg a \wedge \neg b.$$

Ce théorème montre que l'opérateur de conjonction peut être défini à partir des opérateurs de négation et de disjonction à l'aide de la formule : $a \wedge b \equiv \neg(\neg a \vee \neg b)$ et de même, l'opérateur de disjonction peut être défini à partir des opérateurs de négation et de conjonction : $a \vee b \equiv \neg(\neg a \wedge \neg b)$.
Ajoutons le résultat suivant :

Théorème. — *On dispose en outre des équivalences :*

$$(a \Rightarrow b) \equiv (\neg a \vee b) \qquad et \qquad (a \Leftrightarrow b) \equiv \Big((a \Rightarrow b) \wedge (b \Rightarrow a)\Big)$$

Associés au principe de substitution, ces deux théorèmes prouvent que toute formule logique est équivalente à une formule logique définie à l'aide des seuls connecteurs logiques $\neg$ et $\wedge$ (ou $\neg$ et $\vee$).
On dit que $\{\neg, \wedge\}$ et $\{\neg, \vee\}$ sont des *systèmes complets* de connecteurs logiques.

2.3 Formes normales

• Algèbre de Boole

Nous allons dorénavant alléger nos notations en convenant que $\neg a$ sera noté $\overline{a}$, que $a \wedge b$ sera noté ab, que $a \vee b$ sera noté $a + b$ et enfin que l'équivalence $\equiv$ sera représentée par une égalité $=$.
Par exemple, $\neg(a \vee b)$ sera noté $\overline{a + b}$. Ces notations, inspirées du calcul arithmétique, se justifient par la commutativité et l'associativité des opérateurs de conjonction et de disjonction :

$$a + (b + c) = (a + b) + c \quad \text{et} \quad a(bc) = (ab)c$$

ainsi que par la propriété de distributivité :

$$a(b + c) = ab + ac.$$

Il convient néanmoins de rester prudent : certaines équivalences demeurent surprenantes, comme par exemple : $a + bc = (a + b)(a + c)$. En effet :

$$(a + b)(a + c) = a^2 + ab + ac + bc = a + ab + ac + bc = a(1 + b) + ac + bc = a + ac + bc = a(1 + c) + bc = a + bc.$$

Ces égalités étonnantes résultent du fait que ces deux lois *ne confèrent pas* à $\{0,1\}$ une structure d'anneau. En effet, l'addition ainsi définie n'est pas une loi de groupe. Pour retrouver la structure d'anneau de $\mathbb{Z}/2\mathbb{Z}$, il ne faut pas utiliser le « ou » logique mais le « ou exclusif », qu'on note dans ce contexte $\oplus$:

$+$	0	1
0	0	1
1	1	**1**

$\oplus$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

• Forme normales conjonctives et disjonctives

Pour manipuler une formule logique, il est souvent utile de se ramener à une formule logiquement équivalente et de syntaxe plus simple. On peut chercher à minimiser le nombre de connecteurs logiques utilisés, mais ceci est un problème délicat à résoudre, ou chercher à obtenir une expression standardisée. C'est ce deuxième point que nous allons aborder maintenant.

On appelle *littéral* toute formule logique de la forme a ou $\overline{a}$, où a est une variable propositionnelle.
On appelle *disjonction* toute formule logique F s'écrivant $F_1 + \cdots + F_n$, où $F_1, \ldots, F_n$ sont des formules logiques.
On appelle *conjonction* toute formule logique F s'écrivant $F_1 \cdots F_n$, où $F_1, \ldots, F_n$ sont des formules logiques.

Remarque. F est une disjonction si et seulement si $\overline{F}$ est une conjonction.

Théorème. — *Toute formule logique est équivalente à une disjonction de conjonctions de littéraux ; de même, toute formule logique est équivalente à une conjonction de disjonctions de littéraux.*

Preuve. Compte tenu des lois de De Morgan, on peut ne considérer que les formules construites à l'aide des opérateurs de négation et de conjonction.
Montrons par récurrence sur la longueur de F que F est équivalente à une disjonction de conjonctions de littéraux et à une conjonction de disjonctions de littéraux.

- C'est clair si F est un littéral.
- Si $F = \overline{F_1}$, on applique l'hypothèse de récurrence à F_1 et on applique la remarque précédente.
- Si $F = F_1 F_2$, l'hypothèse de récurrence appliquée à F_1 et F_2 montre que F est équivalente à une conjonction de disjonctions de littéraux.
  De plus, si $F_1 \equiv A_1 + \cdots + A_p$ et $F_2 \equiv B_1 + \cdots + B_q$, où $A_1, \ldots, A_p, B_1, \ldots, B_q$ sont des conjonctions de littéraux, alors $F \equiv \sum_{i,j} A_i B_j$, donc F est équivalente à une disjonction de conjonctions de littéraux.

□

Une formule est en *forme normale conjonctive* si elle est composée de conjonctions de disjonctions de littéraux, et en *forme normale disjonctive* si elle est composée de disjonctions de conjonctions de littéraux.
La méthode décrite dans la démonstration pour obtenir une forme normale se révèle peu pratique à l'usage. Nous allons montrer sur un exemple comment la table de vérité associée à la formule logique permet d'obtenir plus facilement cette transformation.
Considérons par exemple la formule logique $F = a \oplus (\neg b \Rightarrow c)$, et sa table de vérité :

a	b	c	$a \oplus (\neg b \Rightarrow c)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Les lignes de la table pour lesquelles F prend la valeur 1 correspondent à : $(a,b,c) = (0,0,1), (0,1,0), (0,1,1), (1,0,0)$, ce qui nous permet d'écrire : $F \equiv \overline{a}\overline{b}c + \overline{a}b\overline{c} + \overline{a}bc + a\overline{b}\overline{c}$.
Dans la formule qu'on obtient par cette technique, chacune des variables propositionnelles (ici a, b, c) apparaît exactement une fois dans chaque conjonction de littéraux, ce qui assure sont unicité à permutation des variables près.

De même, en considérant les lignes de la table pour lesquelles F prend la valeur 0, on obtient la forme normale disjonctive de $\overline{F}$: $\overline{F} \equiv \overline{a}\overline{b}\overline{c} + a\overline{b}c + ab\overline{c} + abc$; en appliquant une loi de DE MORGAN, on en déduit : $F \equiv (a+b+c)(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+c)(\overline{a}+\overline{b}+\overline{c})$; on obtient une *forme normale conjonctive* de F, elle aussi unique à permutation près.

2.4 Tableaux de KARNAUGH

Nous l'avons dit, l'intérêt des formes normales est d'obtenir une formulation standardisée, unique à permutation près. A contrario, les formules obtenues utilisent un nombre important de connecteurs logiques, nombre qui peut parfois être réduit, soit par simplification directe (par exemple, $abc + ab\overline{c}$ est équivalent à ab), soit en utilisant certaines techniques, notamment les tableaux de KARNAUGH.
Il s'agit de tableaux à deux entrées, les valeurs prises par les variables étant disposées en ligne ou en colonne de sorte que deux entrées adjacentes ne diffèrent que d'une valeur logique. Dans l'exemple précédent, un tableau de KARNAUGH associé à F est par exemple :

		bc			
		00	01	11	10
a	0	0	1	1	1
	1	1	0	0	0

Notez bien l'ordre dans lequel sont disposés les différentes valeurs prises par bc : deux cases consécutives ne diffèrent que d'un bit, et c'est aussi le cas de la première et de la dernière, qui seront donc considérées comme des cases adjacentes (d'un point de vue topologique, les tableaux de KARNAUGH sont donc des tores).

Dans un tel tableau, chaque groupe de 8, 4 ou 2 valeurs voisines égales à 1 peut être simplifié. Par exemple, le groupe encadré ci-dessous correspond initialement à l'expression $\overline{a}\overline{b}c + \overline{a}bc$ et peut être simplifié en $\overline{a}c$:

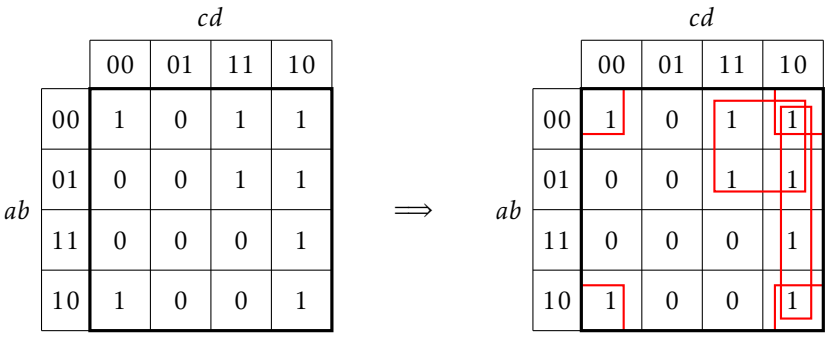
		bc			
		00	01	11	10
a	0	0	1	1	1
	1	1	0	0	0

Pour simplifier l'expression générale, il faut recouvrir tous les 1 présents dans ce tableau par des rectangles de la taille la plus grande possible (plus le rectangle est grand, plus il correspond à une expression simple). Par exemple, le tableau précédent sera recouvert de la façon suivante, ce qui conduit à la formule $F \equiv \overline{a}c + \overline{a}b + a\overline{b}\overline{c}$.

		bc			
		00	01	11	10
a	0	0	1	1	1
	1	1	0	0	0

De même, regrouper les 0 permet d'obtenir une forme disjonctive de $\overline{F}$, et donc une forme conjonctive de F.
Toujours avec l'exemple précédent, $\overline{F} \equiv \overline{a}\overline{b}\overline{c} + ac + ab$, donc $F \equiv (a+b+c)(\overline{a}+\overline{c})(\overline{a}+\overline{b})$.

Un autre exemple, avec cette fois-ci quatre variables. Le tableau suivant conduit aux regroupements indiqués puis à la formule : $F \equiv \overline{a}c + c\overline{d} + \overline{b}\,\overline{d}$.



(Notez en particulier que les quatre angles sont regroupés au sein d'un même rectangle.)

De même, les 0 peuvent être regroupés en trois groupes de quatre, ce qui conduit à : $\overline{F} \equiv b\overline{c} + ad + \overline{c}d$, puis à : $F \equiv (\overline{b} + c)(\overline{a} + \overline{d})(c + \overline{d})$.

3. Évaluation d'une formule logique

Dans cette partie, nous allons nous intéresser à l'évaluation automatique d'une formule logique en fonction des valeurs prises par chacune des variables qui la composent, ce qui nous conduira à la recherche des tautologies et de la satisfiabilité d'une expression logique.

3.1 Représentation en CAML d'une formule logique

De la définition d'une formule logique découle le type *formule* défini de la manière suivante :

```
type formule = Const of bool
             | Var of char
             | Op_unaire of (bool -> bool) * formule
             | Op_binaire of (bool -> bool -> bool) * formule * formule ;;
```

La souplesse de cette définition nous permet de définir aisément les opérateurs logiques dont nous aurons besoin, comme par exemple :

```
let neg p = not p ;;

let et p q = p && q ;;
let ou p q = p || q ;;
let impl p q = q || (not p) ;;
let equiv p q = p = q ;;
let xor p q = not (p = q) ;;
```

J'ai utilisé ici des équivalences logiques pour définir plusieurs de ces opérateurs, mais il aurait été possible de les définir par le biais de leur table de vérité, comme par exemple :

```
let xor = fun
  | false false -> false
  | false true  -> true
  | true false  -> true
  | true true   -> false ;;
```

À titre d'exemple, la formule $F = (a \vee b) \Leftrightarrow (\neg a \wedge c)$ sera représentée par :

```
# let F = Op_binaire(equiv,Op_binaire(ou,Var 'a',Var 'b'),
                           Op_binaire(et,Op_unaire(neg, Var 'a'),Var 'c')) ;;

F: formule = Op_binaire
    (<fun>, Op_binaire (<fun>, Var 'a', Var 'b'),
     Op_binaire (<fun>, Op_unaire (<fun>, Var 'a'), Var 'c'))
```

• Un analyseur lexical

Il s'avère rapidement fastidieux de définir les formules logiques de cette façon, aussi allons-nous supposer désormais posséder un *analyseur lexical*, à savoir une fonction qui décompose une chaîne de caractères en unités lexicales. Par exemple, pour définir la formule logique présentée plus haut, il nous suffira désormais d'écrire :

```
let F = analyseur "(a ou b) <=> (non a et c)" ;;
```

Le module **genlex** propose un analyseur lexical générique que j'ai utilisé pour définir la fonction **analyseur** ; on trouvera sa définition en annexe.

3.2 Évaluation d'une formule logique

Pour pouvoir évaluer une formule logique, il faut qu'à chaque variable soit associée une valeur de vérité. Il est naturel d'utiliser une table d'association, ici représentée par le type *char * bool list*.
Rappelons que la fonction prédéfinie **assoc** retourne la valeur associée à une clé dans une table d'association. Pour mémoire sa définition est la suivante :

```
let rec assoc k = function                    (* prédéfinie en Caml *)
  | []                  -> raise Not_found
  | (x, y)::_ when k = x -> y
  | _::q                -> assoc k q ;;
```

L'évaluation d'une formule pour une liste de valeurs donnée s'écrit alors :

```
let rec evalue f lst = match f with
  | Const c               -> c
  | Var v                 -> assoc v lst
  | Op_unaire (u, f1)     -> u (evalue f1 lst)
  | Op_binaire (b, f1, f2) -> b (evalue f1 lst) (evalue f2 lst) ;;
```

Cette fonction est de type : *formule –> (char * bool) list –> bool*.
Par exemple :

```
# let F = analyseur "(a ou b) <=> (non a et c)"
  in evalue F [('a',true) ; ('b',true) ; ('c',false)] ;;
- : bool = false
```

3.3 Satisfiabilité, tautologie

Nous allons maintenant chercher à reconnaitre automatiquement une tautologie ou une formule satisfiable. Dans un premier temps, nous allons avoir besoin d'une fonction qui extrait la liste des variables d'une formule :

```
let liste_des_variables f =
  let rec cherche_var lst = function
    | Const c               -> lst
    | Var p when (mem p lst) -> lst
    | Var p                 -> p::lst
    | Op_unaire (u, f1)     -> cherche_var lst f1
    | Op_binaire (b, f1, f2) -> cherche_var (cherche_var lst f1) f2
  in cherche_var [] f ;;
```

Voici un exemple d'utilisation de cette fonction :

```
# let F = analyseur "(a ou b) <=> (non a et c)" in liste_des_variables F ;;
- : char list = ['c'; 'b'; 'a']
```

Pour tester une tautologie, nous allons appliquer la méthode naïve : affecter à chacune des variables les deux valeurs vrai et faux, ce qui va nécessiter 2^n évaluations, où n désigne le nombre de variables de la formule.

On procède récursivement de la manière suivante : si la formule ne comporte pas de variable, on l'évalue ; dans le cas contraire, on substitue à la première variable de la liste d'abord la valeur vrai, puis la valeur faux, en procédant à chaque fois à un appel récursif sur la liste obtenue après substitution.

Il est donc nécessaire d'écrire une fonction substituant toute les occurrences d'une variable **p** dans une formule **f** par une constante booléenne **a** :

```
let rec subs f p a = match f with
  | Const c                -> Const c
  | Var q when q = p       -> Const a
  | Var q                  -> Var q
  | Op_unaire (u, f1)      -> Op_unaire (u, (subs f1 p a))
  | Op_binaire (b, f1, f2) -> Op_binaire (b, (subs f1 p a), (subs f2 p a)) ;;
```

Le type de cette fonction est : *formule -> char -> bool -> formule*.

Il est alors enfin possible d'écrire la fonction de vérification des tautologies :

```
exception Echec ;;

let est_une_tautologie f =
  let rec teste f = function
    | []   -> if not (evalue f []) then raise Echec
    | t::q -> let g = subs f t true in teste g q ;
              let g = subs f t false in teste g q
  in let lst = liste_des_variables f
  in try teste f lst ; true
     with Echec -> false ;;
```

Voici à titre d'exemple la justification du raisonnement par contraposée et du raisonnement par l'absurde :

```
# est_une_tautologie (analyseur "(a => b) <=> (non b => non a)") ;;
- : bool = true
# est_une_tautologie (analyseur "(non a => 0) <=> a") ;;
- : bool = true
```

La satisfiabilité se traite de manière analogue, à ceci près qu'on souhaite cette fois obtenir toutes les distributions de vérité affectant la valeur « vrai » à la formule logique.

```
let affiche lst =
  let affiche_valeur = function
    | (c,true)  -> print_char c ; print_string " = vrai  "
    | (c,false) -> print_char c ; print_string " = faux  "
  in do_list affiche_valeur lst ;
  print_newline () ;;

let satisfiabilite f =
  let rec teste f s = function
    | []   -> if evalue f s then affiche s
    | t::q -> teste f ((t,false)::s) q ;
              teste f ((t,true)::s) q
  in let lst = liste_des_variables f
  in teste f [] lst ;;
```

Cherchons par exemple les valeurs à attribuer à a et b pour que $(a \Rightarrow b) \Leftrightarrow (\neg a \Rightarrow \neg b)$ soit vrai :

```
# let F = analyseur "(a => b) <=> (non a => non b)" in satisfiabilite F ;;
a = faux  b = faux
a = vrai  b = vrai
- : unit = ()
```

Exemple. Terminons en résolvant un petit problème de logique élémentaire :

> Vous êtes perdus dans le désert et vous suivez une piste depuis de longues heures quand vous débouchez soudain sur une bifurcation. Vous savez que les deux pistes qui s'ouvrent à vous peuvent éventuellement conduire à une oasis, mais aussi vous perdre à tout jamais. Chacune d'elles est gardée par un sphinx qui s'anime à votre arrivée et commence à parler :
> Le premier vous dit : « une au moins des deux pistes conduit à une oasis. »
> Le second ajoute : « la piste de droite se perd dans le désert. »
> Sachant que les deux sphinx disent tous deux la vérité, ou bien mentent tous deux, que faites vous ?

Considérons les deux assertions a : « la piste de droite conduit à une oasis » et b : « la piste de gauche conduit à une oasis ». Le premier sphinx affirme « $a \vee b$ » et le deuxième « $\neg a$ ».
Posons $\mathrm{F} = \Big((a \vee b) \wedge \neg a\Big) \vee \Big(\neg(a \vee b) \wedge a\Big)$. Par hypothèse, la solution (s'il en existe une) se trouve parmi les distributions qui satisfont F.

```
# let F = analyseur "((a ou b) et non a) ou (non (a ou b) et a)"
  in satisfiabilite F ;;
a = faux  b = vrai
- : unit = ()
```

Il n'y a qu'une seule solution, donc on peut affirmer que la piste de droite se perd dans le désert et que celle de gauche conduit à une oasis.

4. Le problème *n*-SAT

Nous venons d'écrire un algorithme qui résout le problème de la satisfiabilité d'une formule en temps exponentiel. Nous allons montrer qu'il est possible de faire mieux en se restreignant à un ensemble plus réduit de formules logiques.
Nous allons tout d'abord nous restreindre aux formules mises sous forme normale conjonctive. On appelle *clause d'ordre n* toute disjonction d'au plus n littéraux et *forme normale conjonctive d'ordre n* toute conjonction de clauses d'ordre n. Déterminer si une formule sous forme normale conjonctive d'ordre n est satisfiable est appelé *problème n-SAT*.

4.1 Le problème 2-SAT

Le problème 2-SAT revient donc à essayer de satisfaire une conjonction de clauses d'ordre 2, comme par exemple :

$$f_1 = (a+\overline{b})(a+c)(\overline{b}+\overline{c})(b+\overline{c}) \qquad \text{ou} \qquad f_2 = (a+b)(\overline{a}+c)(a+\overline{b})(\overline{b}+c)(\overline{a}+\overline{c})$$

Il existe plusieurs solutions de coût polynomial à ce problème, nous allons présenter l'algorithme de Aspvall, Plass et Tarjan.
Cet algorithme repose sur le lemme suivant :

Lemme. — *La clause d'ordre 2 $(a \vee b)$ est équivalente à la formule $(\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$.*

Preuve. Il suffit de construire la table de vérité de la seconde formule :

a	b	$\neg b \Rightarrow a$	$\neg a \Rightarrow b$	$(\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	1	1

puis observer que c'est la même que pour $a \vee b$. □

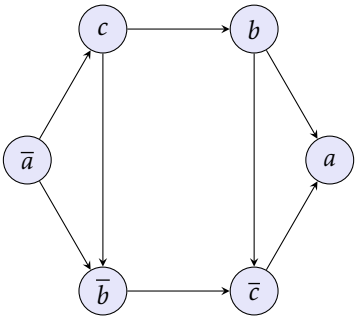
Pour résoudre le problème 2-SAT on construit le graphe orienté $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ en suivant les règles suivantes :

- les sommets V correspondent aux différentes variables propositionnelles a_i présentes dans la formule, ainsi que leurs négations $\overline{a}_i$;
- à chaque clause $(a_i + a_j)$ sont associés les arcs (orientés) $(\overline{a}_i, a_j)$ et $(\overline{a}_j, a_i)$.

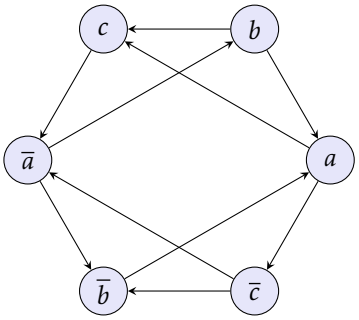
On trouvera figure 2 les graphes G_1 et G_2 associés aux formules f_1 et f_2 définies plus haut.

Théorème. — *Si la formule f est satisfiable, alors pour toute variable v les sommets v et $\overline{v}$ n'appartiennent pas à la même composante fortement connexe du graphe.*

Rappelons que deux sommets s_1 et s_2 appartiennent à la même composante fortement connexe s'il existe un chemin de s_1 à s_2 et un chemin de s_2 à s_1.



$f_1 = (a+\overline{b})(a+c)(\overline{b}+\overline{c})(b+\overline{c})$



$f_2 = (a+b)(\overline{a}+c)(a+\overline{b})(\overline{b}+c)(\overline{a}+\overline{c})$

FIGURE 2 – Les graphes associés aux formules f_1 et f_2.

Preuve. Si la formule f est satisfiable, il existe une distribution de vérité pour laquelle les implications logiques représentées par les arcs du graphe G sont satisfaites. Si deux sommets s_1 et s_2 appartiennent à la même composante fortement connexe, la formule $s_1 \Leftrightarrow s_2$ doit donc être satisfaite, ce qui impose à cette distribution d'être constante sur chacune des composantes fortement connexes du graphe. Ainsi, v et $\overline{v}$ ne peuvent appartenir à la même composante connexe. □

Exemple. f_2 ne peut être satisfaite puisque a et $\overline{a}$ appartiennent à la même composante fortement connexe.

Nous allons maintenant prouver la réciproque de ce résultat en montrant comment construire une distribution de vérité qui satisfait f lorsque pour tout sommet v, le sommet $\overline{v}$ n'appartient pas à la même composante connexe que v. Pour cela, on construit le graphe orienté H de la manière suivante :

- les sommets de H sont les composantes fortement connexes de G ;
- il existe un arc allant d'une composante X à une composante Y dans H s'il existe un arc allant d'un sommet de X à un sommet de Y dans G.

LEMME. — H *est un graphe sans circuit.*

Preuve. Soient X et Y deux composantes fortement connexes de G distinctes. S'il existe un chemin dans H reliant X à Y, il existe un chemin dans G reliant un sommet $s_1 \in$ X à un sommet $s_2 \in$ Y. De même, s'il existe un chemin dans H reliant Y à X, il existe un chemin dans G reliant un sommet $s_3 \in$ Y à un sommet $s_4 \in$ X. Mais s_1 et s_4 appartiennent à la même composante connexe X donc il existe un chemin dans G menant de s_4 à s_1 et de même un chemin dans G menant de s_2 à s_3. Ceci montre l'existence d'un cycle $s_1 \to s_2 \to s_3 \to s_4 \to s_1$ qui prouve que ces quatre sommets de G doivent appartenir à la même composante connexe, ce qui est absurde. □

L'absence de circuit permet d'établir sur l'ensemble des sommets de H un ordre partiel en convenant que X $\preccurlyeq$ Y si et seulement si X = Y ou s'il existe un arc reliant X à Y dans H. Nous allons compléter cet ordre partiel pour obtenir un ordre total en appliquant à H l'*algorithme de tri topologique*, déjà étudié au cours du chapitre consacré aux graphes : il s'agit d'effectuer un parcours en profondeur en insérant en guise de traitement les sommets en tête d'une liste chaînée.

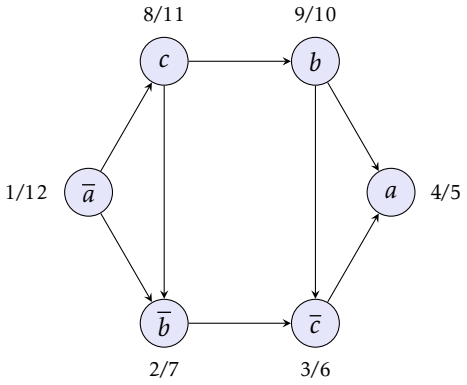
Examinons le cas de la formule f_1. Chacun des six sommets constitue à lui seul une composante fortement connexe donc les graphes G et H coïncident. Dans la figure 3 on a placé à côté de chaque sommet deux entiers : le date d'entrée dans la pile « àTraiter » associée au parcours en profondeur et la date d'entrée dans la liste chaînée « ordre ».

LEMME. — *Si* X $\preccurlyeq$ Y *alors* X *est situé avant* Y *dans la liste chaînée.*

Preuve. Considérons le moment où on explore l'arc X → Y. Si Y n'a pas encore été vu il rentrera dans la liste chaînée avant X et sera donc situé après X quand X y rentrera à son tour (puisque l'insertion se fait en tête de liste).
Si Y a déjà été vu, deux cas sont envisageables : si Y est déjà dans la liste chaînée, le problème est réglé : au moment où X y entrera à son tour il sera bien situé avant Y dans cette liste.

```
procedure TRI_TOPOLOGIQUE(graphe : H)
    ordre ← [ ]
    déjàVus ← ∅
    while |ordre| < |H| do
        H \ ordre → s_0
        DFS(s_0)
    return ordre

procedure DFS(sommet : s_0)
    àTraiter ← s_0
    déjàVus ← s_0
    while àTraiter ≠ ∅ do
        àTraiter → s
        for t ∈ voisins(s) do
            if t ∉ déjàVus then
                DFS(t)
        ordre ← s :: ordre
```



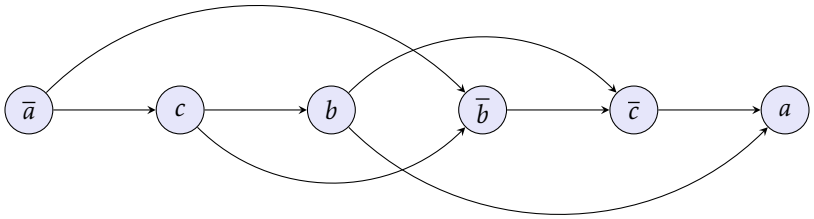


FIGURE 3 – Tri topologique du graphe associé à f_1. On notera que lorsque le graphe est dessiné suivant l'ordre topologique les arcs sont tous orientés de gauche à droite.

Il reste à examiner le cas où Y a déjà été vu mais se trouve encore dans la pile. Mais cela signifie que X est un descendant de Y avec pour conséquence l'existence d'un cycle dans H, ce qui est exclus d'après le lemme précédent. □

On dispose donc désormais d'un ordre total $\leqslant$ sur les composantes fortement connexes de G qui prolonge l'ordre partiel $\preccurlyeq$. On définit alors la distribution de vérité sur f de la manière suivante : si s est dans la composante X et $\overline{s}$ dans la composante Y on pose $s = 0$ si $X < Y$ et $s = 1$ si $Y < X$.

Exemple. Dans le cas de la formule f_1 cela revient à poser $a = 1$, $b = 0$, $c = 0$ et c'est en effet une distribution qui satisfait f_1.

THÉORÈME. — *Ainsi définie, cette distribution de vérité satisfait f.*

Preuve. Considérons deux sommets s_1 et s_2 de G reliés par un arc $s_1 \to s_2$. Il faut montrer que pour la distribution de vérité définie ci-dessus l'implication $s_1 \Rightarrow s_2$ est vraie. Notons tout de suite que par construction il existe aussi un arc $\overline{s_2} \to \overline{s_1}$.
Notons X_1 la composante fortement connexe de s_1 et X_2 celle de s_2. On a $X_1 \leqslant X_2$.
Notons Y_1 la composante fortement connexe de $\overline{s_1}$ et Y_2 celle de $\overline{s_2}$. On a $Y_2 \leqslant Y_1$.
Il reste à constater que le seul cas ou $s_1 \Rightarrow s_2$ pourrait être faux serait d'avoir $s_1 = 1$ et $s_2 = 0$ avec pour conséquence $Y_1 < X_1$ et $X_2 < Y_2$, mais ces inégalités sont incompatibles avec les précédentes. □

Complexité du problème 2-SAT
Nous n'avons pas détaillé la manière d'obtenir les composantes fortement connexes d'un graphe orienté, mais il existe des algorithmes dont le coût est un $\Theta(|V| + |E|)$ (l'algorithme de TARJAN par exemple). Si n désigne le nombre de variables de la formule à satisfaire nous avons $|V| \leqslant 2n$ et $|E| \leqslant |V|(|V| - 1)$ donc le calcul des composantes fortement connexes de G est un $O(n^2)$.
Le tri topologique du graphe H consiste à effectuer un parcours en profondeur donc son coût est lui aussi un $O(|V| + |E|) = O(n^2)$.
L'attribution des valeurs de vérité peut aisément se faire pour un coût en $O(n^2)$ (déterminer si $X < Y$ est de coût linéaire dans la liste chaînée), donc on peut affirmer qu'il existe une solution de coût polynomial au problème 2-SAT.

4.2 Le problème 3-SAT

Le problème 3-SAT est particulièrement important car tous les problèmes n-SAT pour $n \geqslant 3$ en découlent. En effet, toute clause $(x_1 + x_2 + \cdots + x_n)$ avec $n \geqslant 3$ peut se mettre sous la forme équivalente :

$$(x_1 + x_2 + y_2)(\overline{y}_2 + x_3 + y_3)(\overline{y}_3 + x_4 + y_4)\cdots(\overline{y}_{n-3} + x_{n-2} + y_{n-2})(\overline{y}_{n-2} + x_{n-1} + x_n)$$

ce qui permet de réduire en temps polynomial un problème n-SAT pour $n \geqslant 3$ à un problème 3-SAT.
Malheureusement, on ne connait aucun algorithme de coût polynomial pour le résoudre ; il s'agit en effet d'un problème *NP-complet*.

• NP-complétude

En théorie de la complexité, on classe les problèmes de décision en plusieurs familles, en particulier :

- la classe P constituée des problèmes de décision qui peuvent être résolus en temps polynomial par rapport à la taille de l'entrée ;
- la classe NP constituée des problèmes de décision pour lesquels on peut vérifier la validité d'une solution en temps polynomial.

Nous venons par exemple de prouver que le problème 2-SAT appartient à la classe P, et il n'est pas bien difficile de constater que les problèmes n-SAT appartiennent à la classe NP : étant donné une distribution de vérité, l'évaluation de la formule logique que nous avons programmée à la section 3.2 s'exécute en temps polynomial.

Il est évident que nous disposons de l'inclusion $\mathrm{P} \subset \mathrm{NP}$; Le problème de l'inclusion réciproque est un des plus grands défis de l'informatique théorique et à ce titre figure dans la liste des problèmes du millénaire.

Parmi les problèmes de la classe NP, on distingue certains d'entre eux, qualifiés de problèmes *NP-complets*. D'une certaine façon, ce sont les plus difficiles des problèmes NP : si on connaissait une solution polynomiale pour l'un d'eux, on en connaîtrait une pour tout problème de la classe, et on aurait l'égalité P = NP. Le problème SAT a la particularité d'avoir été le premier problème NP-complet trouvé ; ce résultat est maintenant connu sous le nom de théorème de Cook. De nombreux autres problèmes ont ensuite été rajoutés à la famille des problèmes NP-complets en procédant par réduction polynomiale, comme nous l'avons fait pour le problème 3-SAT.

5. Exercices

Exercice 1 Le NAND, appelé aussi le connecteur de Sheffer, est défini par : $a \mid b = \neg(a \wedge b)$, a et b désignant deux variables propositionnelles.
Construire les tables de vérité de $a \mid b$ et de $a \mid a$, et en déduire l'existence de formules ne contenant que les variables a et b et des occurrences du connecteur de Sheffer et qui sont logiquement équivalentes à : $\neg a$, $a \wedge b$, $a \vee b$, $a \Rightarrow b$, $a \Leftrightarrow b$.
Nous venons de démontrer que le connecteur de Sheffer est à lui seul un système complet. Montrer qu'il en est de même de l'opérateur NOR (non ou).

Exercice 2 Sans utiliser de table de vérité, simplifier l'expression logique suivante : $\neg(a \wedge b) \wedge (a \vee \neg b) \wedge (a \vee b)$.

Exercice 3 Démontrer que les propositions suivantes sont des tautologies, sans utiliser de table de vérité :

$$ab + c + \overline{b}\overline{c} + \overline{a}\,\overline{c} \qquad \text{et} \qquad a + \overline{b}\overline{c} + \overline{a}c + b\overline{c}.$$

Exercice 4 Mettre la proposition logique : $(\neg a \vee b) \wedge c \Leftrightarrow a \oplus c$ sous forme normale disjonctive puis sous forme normale conjonctive. À l'aide des tableaux de Karnaugh, simplifier ensuite les expressions obtenues.

Exercice 5 a, b, c et d désignant des variables booléennes, on considère les formules logiques F et G définies par :

F = 1 si et seulement si $a + b \leqslant c + d$ (il s'agit ici de l'addition usuelle) ;

G = 1 si et seulement si l'entier dont l'écriture en base 2 est $abcd$ est strictement inférieur à 10.

À l'aide de tableaux de Karnaugh, donner une expression simple de F et de G.

Exercice 6 Un coffre-fort est muni de n serrures et peut être ouvert uniquement lorsque ces n serrures sont simultanément ouvertes. Cinq personnes, nommées A, B, C, D, E, doivent recevoir des clés correspondant à certaines serrures. Chaque clé peut être disponible en autant d'exemplaires qu'on le souhaite. On demande de choisir pour l'entier n la plus petite valeur possible, et de lui associer une répartition des clés entre les cinq personnes, de telle manière que le coffre puisse être ouvert si et seulement si on se trouve dans une au moins des situations suivantes :

- présence simultanée de A et B ;
- présence simultanée de A, C et D ;
- présence simultanée de B, D et E.

On désigne par a l'assertion : « A est présent », et on définit de même les assertions b, c, d, e.

a) Exprimer par une formule logique F dépendant des variables a, b, c, d, e la condition pour que le coffre puisse être ouvert.

b) Construire le tableau de Karnaugh de F, et en déduire une expression de F sous forme conjonctive.

c) En déduire une valeur de n et une répartition adéquate des clés, en cherchant à en minimiser le nombre.

Exercice 7 On travaille avec un langage de programmation qui ne dispose que d'une unique instruction : $a \leftarrow b \oplus c$ où a, b, c sont des variables pouvant contenir des entiers codés sur n bits. L'exécution de cette instruction se déroule ainsi : soit $b_{n-1}\cdots b_1 b_0$ la représentation binaire de l'entier contenu dans la variable b et $c_{n-1}\cdots c_1 c_0$ celle de l'entier contenu dans la variable c ; alors l'instruction range dans la variable a l'entier dont la représentation binaire $a_{n-1}\cdots a_1 a_0$ est définie par : $\forall k \in [\![0, n-1]\!]$, $a_k = b_k \oplus c_k$.
Soient u et v deux variables. Donner une suite d'instructions à exécuter pour échanger les valeurs contenues dans les variables u et v sans utiliser d'autre variable.

Exercice 8 Laquelle de ces deux formules est une tautologie ?

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

(Il s'agit de la loi de Peirce.)

Exercice 9 On interroge un logicien qui dit toujours la vérité sur sa vie sentimentale. À la question : « est-il vrai que si vous aimez Marie, alors vous aimez Anne ? » il répond :

- si c'est vrai, alors j'aime Marie ;
- si j'aime Marie, alors c'est vrai.

Qu'en concluez-vous ?

Exercice 10 Voici le curieux règlement d'un club britannique.

article 1 : tout membre non écossais porte des chaussures oranges ;

article 2 : tout membre porte une jupe ou ne porte pas de chaussures oranges ;

article 3 : les membres mariés ne sortent pas le dimanche ;

article 4 : un membre sort le dimanche si et seulement s'il est écossais ;

article 5 : tout membre qui porte une jupe est écossais et marié ;

article 6 : tout membre écossais porte une jupe.

À votre avis, ce règlement est-il cohérent ?

Annexe : définition d'un analyseur lexical

Le module **genlex** de la distribution standard de Caml Light permet de définir un analyseur lexical simple [2].
La fonction **make_lexer** prend en argument la liste des mots-clés et retourne la fonction d'analyse lexicale proprement dite :

```
#open "genlex" ;;

let lexer = make_lexer ["non"; "ou"; "et"; "=>"; "<=>"; "xor"; "("; ")"] ;;
```

Ceci définit une fonction **lexer** de type *char stream –> token stream* qui prend en argument un flux [3] de caractères et produit un flux de lexèmes en sortie.
Il reste à convertir ce flux de lexèmes en une formule logique à l'aide de la fonction :

```
let rec parse = function
  | [< 'Int 0 ; (parse_bin (Const false)) n >] -> n
  | [< 'Int 1 ; (parse_bin (Const true)) n >] -> n
  | [< 'Ident c ; (parse_bin (Var (nth_char c 0))) n >] -> n
  | [< 'Kwd "(" ; parse n1 ; 'Kwd ")" ; (parse_bin n1) n2 >] -> n2
  | [< 'Kwd "non" ; parse_uni n1 ; (parse_bin (Op_unaire (neg, n1))) n2 >] -> n2
and parse_uni = function
  | [< 'Int 0 >] -> Const false
  | [< 'Int 1 >] -> Const true
  | [< 'Ident c >] -> Var (nth_char c 0)
  | [< 'Kwd "(" ; parse n ; 'Kwd ")" >] -> n
and parse_bin n1 = function
  | [< 'Kwd "et" ; parse n2 >] -> Op_binaire (et, n1, n2)
  | [< 'Kwd "ou" ; parse n2 >] -> Op_binaire (ou, n1, n2)
  | [< 'Kwd "=>" ; parse n2 >] -> Op_binaire (impl, n1, n2)
  | [< 'Kwd "<=>" ; parse n2 >] -> Op_binaire (equiv, n1, n2)
  | [< 'Kwd "xor" ; parse n2 >] -> Op_binaire (xor, n1, n2)
  | [< >] -> n1 ;;

let analyseur s = parse (lexer (stream_of_string s)) ;;
```

♦ ♦
♦

2. Se référer au manuel en ligne pour plus d'information.
3. Tout comme les listes, un *flux* (*stream* en anglais) est une suite de valeurs de même type, mais à la différence des listes l'accès dans un flux est destructif : lorsqu'on consulte le premier élément d'un flux celui-ci est aussitôt retiré et remplacé par l'élément suivant. En outre, l'évaluation d'un flux est paresseuse : les éléments ne sont évalués qu'au moment où on y accède.