

Chapitre 14

LOGIQUE

Notions	Commentaires
Variables propositionnelles, connecteurs logiques, arité. Formules propositionnelles, définition par induction, représentation comme un arbre. Sous-formule. Taille et hauteur d'une formule.	Notations : $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$. Les formules sont des données informatiques. On fait le lien entre les écritures d'une formule comme mot et les parcours d'arbres.
Quantificateurs universel et existentiel. Variables liées, variables libres, portée. Substitution d'une variable.	On ne soulève aucune difficulté technique sur la substitution. L'unification est hors programme.

Extrait de la section 4.5 du programme officiel de MP2I : « Syntaxe des formules logiques ».

Notions	Commentaires
Valuations, valeurs de vérité d'une formule propositionnelle. Satisfiabilité, modèle, ensemble de modèles, tautologie, antilogie. Équivalence sur les formules. Conséquence logique entre deux formules.	Notations V pour la valeur vraie, F pour la valeur fausse. Une formule est satisfiable si elle admet un modèle, tautologique si toute valuation en est un modèle. On peut être amené à ajouter à la syntaxe une formule tautologique et une formule antilogique ; elles sont en ce cas notées \top et \perp . On présente les lois de De Morgan, le tiers exclu et la décomposition de l'implication.
Forme normale conjonctive, forme normale disjonctive. Mise sous forme normale.	On étend la notion à celle de conséquence ϕ d'un ensemble de formules Γ : on note $\Gamma \models \phi$. La compacité est hors programme. Lien entre forme normale disjonctive complète et table de vérité. On peut représenter les formes normales comme des listes de listes de littéraux. Exemple de formule dont la taille des formes normales est exponentiellement plus grande.
Problème SAT, n -SAT, algorithme de Quine.	

Extrait de la section 4.6 du programme officiel de MP2I : « Sémantique de vérité du calcul propositionnel ».

SOMMAIRE

0. Syntaxe des formules logiques	319
0. Formules propositionnelles	319
1. Notations et parcours	320
2. Égalités et substitutions	321

1. Sémantique des formules propositionnelles	323
0. Évaluation d'une proposition logique	323
1. Conséquence, équivalence logique	325
2. Satisfiabilité	327
2. Formes normales.....	329
0. Définitions	329
1. Construction de DNF/CNF	330
2. Représentation informatique d'une formule	332
3. SAT et algorithme de Quine	333
0. Problème SAT	333
1. Algorithme de Quine	334
2. Une petite parenthèse : les graphes de flots	336
4. Quantificateurs et introduction à la logique du premier ordre	337

Le but de l'étude de la logique est de formaliser ce qu'est un raisonnement, et ce qu'est une preuve (MPI). *Il faut donc bien distinguer le raisonnement "méta" que l'on utilise pour formaliser et manipuler la logique, de celui qu'on définit dans le cadre de la logique (et qui est censé décrire le premier.)*

Cela a au moins deux grands intérêts :

- Garantir que ce que l'on prouve est correct, autrement dit garantir que l'on peut correctement déduire la véracité de la conclusion à partir de la véracité du résultat.
- Automatiser la recherche de preuve, ou la vérification de la correction d'une preuve.

En MP2I, on se concentrera surtout sur la formalisation de ce qu'est une *formule propositionnelle*, et sur les liens logiques entre différentes propositions.

En MPI, vous verrez comment prouver forme qu'une proposition est vraie (d'une manière plus efficace et intuitive que le retour sur trace). Ce faisant, vous formaliserez en fait ce qu'est une *preuve* !

0 Syntaxe des formules logiques

0.0 Formules propositionnelles

Intuitivement, une formule propositionnelle est « une propriété mathématique sans quantificateurs ». Formalisons cette notion par induction :

Définition 1 (Formules propositionnelles).

Soit \mathcal{V} un ensemble infini dénombrable dont les éléments sont appelés variables propositionnelles. On notera souvent ces variables x, y, \dots ou encore x_0, x_1, \dots .

L'ensemble \mathcal{P} des **formules propositionnelles** est défini inductivement par :

- $\top, \perp \in \mathcal{P}$
- $\mathcal{V} \subseteq \mathcal{P}$
- si $P, Q \in \mathcal{P}$, alors :
 - $\neg P \in \mathcal{P}$
 - $P \wedge Q \in \mathcal{P}$
 - $P \vee Q \in \mathcal{P}$
 - $P \rightarrow Q \in \mathcal{P}$
 - $P \leftrightarrow Q \in \mathcal{P}$

On note souvent les propriétés logiques P, Q, \dots .

Les symboles utilisés ci-dessus sont :

- \top, \perp (**top, bot (ou bottom)**) : constantes.
- \neg : **non** - négation : connecteur unaire.
- \wedge : **et** - conjonction : connecteur binaire.
- \vee : **ou** - disjonction : connecteur binaire.
- \rightarrow : **implique** - implication : connecteur binaire.
- \leftrightarrow : **équivalent à** - équivalence : connecteur binaire.

Remarque.

- On peut aussi choisir de ne pas mettre les constantes \top et \perp dans la syntaxe des formules propositionnelles, car on pourra en retrouver des formulations équivalentes en utilisant uniquement les autres symboles.
- On appelle aussi une formule propositionnelle une **proposition**. Dans ce contexte, une proposition n'a pas de quantificateurs (ce ne sont pas les "propositions" que vous écrivez tout le temps dans vos cours de maths)!
- ⚠ On ne dit pas pour l'instant ce que *veut dire* $\neg P$, ou encore $P \vee Q$. Les connecteurs n'ont *pas de sens en soi*, ce sont juste des symboles. Nous avons uniquement donné les *règles d'écritures* qui permettent d'écrire une formule propositionnelle, mais n'avons rien dit de comment *comprendre* une telle formule.
Autrement dit, nous avons spécifiée la **syntaxe** mais pas la **sémantique**.
- Nous avons fait pareil avec les types abstraits. La signature d'un type abstrait définit uniquement sa syntaxe ; et nous avons vu qu'il fallait ensuite l'accompagner d'axiomes, de spécifications des fonctions, pour donner du sens.

Proposition 2.

De manière équivalente, on peut représenter une formule par un arbre dont les noeuds internes sont étiquetés par des connecteurs, et les feuilles par des variables propositionnelles ou des constantes. Un noeud étiqueté par un connecteur unaire est d'arité 1, binaire d'arité 2.

Exemple.

On pourrait formaliser la notion d'arbre associé à une formule et prouver une bijection entre les arbres et les formules¹. Toutefois, cela n'a que peu d'intérêt logique² puisqu'il s'agit simplement d'interpréter la définition inductive de proposition comme une définition d'arbre étiqueté par les connecteurs.

Type des formules propositionnelles (aussi appelées "propositions") en OCaml :

```
1 type prop =
2   | Const of bool (*top ou bot*)
3   | Var of int (*l'entier i représente la variable x_i*)
4   | Non of prop
5   | Et of prop * prop
6   | Ou of prop * prop
7   | Implique of prop * prop
8   | Equivaut of prop * prop
```



Remarque. on reconnaît une définition d'arbre étiquetés par les constructeurs, et c'est normal !

Exemple. avec ce type OCaml, l'arbre précédent s'écrirait :

Définition 3.

Soit $P \in \mathcal{P}$ une formule propositionnelle. On appelle :

- hauteur de P la hauteur de l'arbre associé à P .
- taille de P le nombre noeuds (internes ou non) de l'arbre.
- sous-formule de P un sous-arbre de P (vu comme une formule).

0.1 Notations et parcours

À quel parcours de l'arbre correspond la formule propositionnelle donnée en exemple ? **Réponse :**

Et dans le langage OCaml ? **Réponse :**

Écrire cette formule sous forme postfixe. **Réponse :**

La notation usuelle des formules propositionnelles est la notation infixe. Il faut donc des parenthèses pour lever les ambiguïtés. Sans parenthèses ni règle d'écriture, une écriture sous forme infixe pourrait correspondre à plusieurs propositions différentes.

1. Vous pouvez le faire en exercice si vous voulez.

2. Mais a des intérêts en tant qu'exercice : vous faire manipuler les définitions et les inductions.

Exemple. $x \wedge y \vee z \wedge a$ peut correspondre à :

Remarque.

- Si on connaît l'arité de chaque noeud (c'est le cas ici), les écritures préfixes et postfixes correspondent à un unique arbre possible (on a une bijection). Il n'y aurait donc pas d'ambiguïté, même sans parenthèses. Ce résultat est connu sous le nom de « lemme de lecture unique »³.
- Malgré cela, on privilégie généralement la notation infixe, munie de règles de priorité et éventuellement de parenthèses (comme en maths !). C'est la version la plus lisible, même si elle a besoin de parenthèses.

Définition 4 (Priorités logiques).

On utilise les règles de priorités (donc de parenthésages implicites) suivantes pour les connecteurs :

- \neg est prioritaire sur \wedge et \vee
- \wedge est prioritaire sur \vee .

Remarque. Il est fortement recommandé d'écrire quand même les parenthèses entre les \vee et les \wedge , par souci de clarté. **Et je vous demande de le faire.**

Exemple. Donner l'arbre associé à $(\neg\neg x \wedge y) \vee (\neg z \wedge x)$

Notations : On écrit parfois :

- \bar{x} pour $\neg x$
- $x.y$ pour $x \wedge y$
- $x + y$ pour $x \vee y$

Cependant, ces notations sont souvent réservées à des variables propositionnelles, et je ne vous les recommande vraiment pas, notamment pour des raisons de distributivité qu'on verra plus tard (le "+" sera distributif sur le "×" en plus de l'inverse... ce qui est très perturbant!).

0.2 Égalités et substitutions

Définition 5 (Égalité syntaxique).

On dit que deux propositions P et Q sont **syntactiquement égales** si leurs arbres associés sont égaux, i.e. si « elles s'écrivent pareil ».

Remarque.

- Le cas des parenthèses est fourbe, car il mène à des écritures « différentes » alors qu'il s'agit de la même formule et du même arbre. Par exemple : $((x \wedge y) \vee z)$ et $(x \wedge y) \vee z$ sont syntactiquement égaux, bien qu'ils semblent avoir une écriture « différente ».
- Ces parenthèses sont la seule difficulté de cette définition.
- Attention à ne pas aller trop vite : ceci est une égalité syntaxique et non sémantique. Autrement dit, $P \rightarrow Q \wedge Q \rightarrow P$ n'est pas syntactiquement égal à $P \leftrightarrow Q$. Notre intuition nous dit que ces formules ont la même *signification*, mais ce sont bien deux formules distinctes syntactiquement.

3. On l'a mentionné rapidement dans la partie hors-programme du cours sur les arbres.

Définition 6 (Extension de l'égalité syntaxique).

On étend l'égalité syntaxique en lui ajoutant les règles suivantes :

- \wedge est commutatif syntaxiquement : $P \wedge Q$ est syntaxiquement égal à $Q \wedge P$. De même pour \vee et \leftrightarrow .
- \wedge est associatif syntaxiquement : $P \wedge (Q \wedge R)$ est syntaxiquement égal à $(P \wedge Q) \wedge R$. De même pour \vee et \leftrightarrow .

Autrement dit, on considère que les enfants d'un noeud de l'arbre ne sont pas ordonnés, et on considère des noeuds \vee et \wedge d'arité quelconque. On notera $\bigvee_{i=0}^n P_i$ pour $P_0 \vee P_1 \vee \dots \vee P_n$. De même pour \wedge .

Exemple.

Remarque. L'égalité syntaxe non-étendue est simple à tester (on parcourt les deux arbres pour vérifier leur égalité). L'égalité syntaxique étendue par associativité et commutativité est plus difficile à tester⁴.

Définition 7 (Substitution).

Soient $P, Q \in \mathcal{P}$, soit $x \in \mathcal{V}$.

La **substitution de x par Q dans P** , notée $P[Q/x]$, est la formule obtenue en remplaçant dans P toutes les occurrences de x par Q .

Formellement, on la définit inductivement :

- $\perp[Q/x] = \perp$, $\top[Q/x] = \top$
- $x[Q/x] = Q$
- si $y \in \mathcal{V} \setminus \{x\}$, $y[Q/x] = y$
- $(\neg P)[Q/x] = \neg(P[Q/x])$
- $(P_1 \wedge P_2)[Q/x] = (P_1[Q/x]) \wedge (P_2[Q/x])$. De même pour \vee , \rightarrow et \leftrightarrow

Remarque.

- Vision arborescente : substituer x par Q revient à remplacer toutes les feuilles étiquetées par x par une copie de l'arbre de Q .
- En toute généralité, la substitution n'est pas commutative : $P[Q/x][R/y] \neq P[R/y][Q/x]$.

Exercice. Trouver une condition nécessaire pour qu'elle le soit. **Réponse :**

4. Et ne parlons pas d'une égalité sémantique : vérifier l'égalité sémantique de formules quantifiées est très littéralement un métier que l'on nomme mathématicien.

1 Sémantique des formules propositionnelles

Nous allons maintenant donner du *sens* aux règles d'écritures définies précédemment.

1.0 Évaluation d'une proposition logique


Dans cette partie, on note $\mathbb{B} = \{\text{Vrai}; \text{Faux}\}$ (abrégiés en V et F) ou de manière équivalente $\mathbb{B} = \{1; 0\}$, l'ensemble des booléens.

Définition 8 (Fonctions logiques).

On définit les fonctions *Non*, *Et*, *Ou*, *Implique*, *Equivaut* par leurs tables de vérité :

x	y	Non(x)	Et(x,y)	Ou(x,y)	Implique(x,y)	Equivaut(x,y)
V	V					
V	F					
F	V					
F	F					

Remarque.

-  Ne pas confondre ces **fonctions** avec les **symboles** de connecteurs logiques $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$.
- *Aide à l'intuition* Notez bien que $\text{Implique}(F, F) = V$, autrement dit "faux implique faux" est vrai. Si ça peut paraître contre-intuitif au départ, c'est pourtant bien le cas, et on veut que ce soit le cas. Voici un exemple pour vous en convaincre :

Il vous semblera tout à fait normal (et même évident) de dire que : si n est un multiple de 4, alors n est pair est **vrai** sur l'ensemble des entiers.

Mais alors, vous conviendrez que vous acceptez comme vraies les propositions suivantes :

- Si 4 est un multiple de 4, alors 4 est pair.
- Si 2 est un multiple de 4, alors 2 est pair.
- Si 1 est un multiple de 4, alors 1 est pair.

On voit ici grâce à ce "théorème" les 3 cas possibles de l'implication (qui rendent l'implication vraie, donc) :

- Vrai implique vrai.
- Faux implique vrai.
- Faux implique faux.

Définition 9 (Valuation).

Une **valuation** est une application de l'ensemble des variables propositionnelles \mathcal{V} dans \mathbb{B} . On la note souvent $v : \mathcal{V} \rightarrow \mathbb{B}$

Remarque. On parle aussi de distribution de vérité. Cela revient à choisir pour chaque variable si elle est vraie ou fausse.

Définition 10 (Modèle (MP2I)).

Une **interprétation**, aussi appelée un **modèle**, d'une formule propositionnelle est une valuation des variables propositionnelles de cette formule.

Remarque. Cette définition de modèle est un cas restreint d'une définition plus large que vous verrez en MPI.

Exemple. Si l'on se restreint à 3 variables x, y, z , alors la fonction v suivante est un modèle de la formule $x \vee y \wedge z$:

$$\begin{aligned} v : \mathcal{V} &\rightarrow \mathbf{B} \\ x &\mapsto F \\ y &\mapsto V \\ z &\mapsto V \end{aligned}$$

Définition 11 (Évaluation).

Soit v une valuation. On définit $eval_v : \mathcal{P} \rightarrow \mathbf{B}$ l'application qui à une formule P associe son évaluation par la valuation v ainsi :

- $eval_v(\top) = V, \quad eval_v(\perp) = F$
- Pour tout $x \in \mathcal{V}, eval_v(x) = v(x)$
- $eval_v(\neg P) = Non(eval_v(P))$
- $eval_v(P \wedge Q) = Et(eval_v(P), eval_v(Q)).$
- $eval_v(P \vee Q) = Ou(eval_v(P), eval_v(Q)).$
- $eval_v(P \rightarrow Q) = Implique(eval_v(P), eval_v(Q)).$
- $eval_v(P \leftrightarrow Q) = Equivaut(eval_v(P), eval_v(Q)).$

Exemple. l'évaluation de la proposition exemple par la valuation précédente donne :

Remarque.

- On note aussi e_v pour $eval_v$, et on parle d'évaluation de P **dans le contexte de v** .
- \mathcal{V} est infini : impossible à programmer, pas pratique pour les exemples. On se limite en général à définir une valuation pour les variables qui apparaissent dans les formules que l'on considère. C'est d'autant plus courant lorsque l'on décrit un modèle d'une formule (comme dans l'exemple précédent de modèle).
- On note $\mathcal{V}(P)$ l'ensemble des variables apparaissant dans P . C'est un ensemble fini puisqu'une formule l'est.
- Si *Vrai*, *Faux* = 1, 0 et plus généralement si $V > F$, alors l'évaluation d'un \wedge est un *min* et celle de \vee un *max*.

Exemple. Vous parlez à trois personnes A, B et C. On note x_A le fait que A dit la vérité, x_B B dit la vérité, x_C C dit la vérité. On peut alors écrire E , une formule représentant le fait qu'au moins l'un d'entre eux dit la vérité. **Réponse :**

Une valuation est ici une façon de répartir qui dit la vérité et qui ment. Si on veut satisfaire la proposition, i.e. assurer qu'au moins une personne dit la vérité, alors voici les valuations qui conviennent :

(Réponse :)

Définition 12 (Satisfaction).

On dit qu'une valuation v satisfait une formule P si $eval_v(P) = Vrai$.

Exemple. Les valuations indiquées à l'exemple précédent satisfont E . On dit aussi qu'elles **valident** E .

1.1 Conséquence, équivalence logique

Maintenant que l'on a de la sémantique, on peut créer des relations entre différentes formules, au delà de la seule égalité syntaxique.

Définition 13 (Conséquence, équivalence).

Soient $P, Q \in \mathcal{P}$ deux formules propositionnelles.

- On dit que Q est une **conséquence logique** de P , et on note $P \models Q$, si toute valuation satisfaisant P satisfait Q .
- Si $\Gamma = \{P_1, \dots, P_n\}$ est un ensemble de formules propositionnelles, on dit que $\Gamma \models Q$ si toute valuation satisfaisant toutes les formules de Γ satisfait également Q . On écrit $\models P$ pour $\emptyset \models P$.
- P et Q sont dits **sémantiquement équivalentes** si $P \models Q$ et $Q \models P$. On note $P \equiv Q$.

\models se prononce « satisfait ».

Exemple :

Remarque.

- Autrement dit, deux formules sont sémantiquement équivalentes si et seulement si elles sont satisfaites par exactement les mêmes valuations.
- Si $P \equiv Q$, on dit aussi que P et Q sont **logiquement équivalentes** ou **tautologiquement équivalentes**.
- \triangle L'équivalence logique n'est **pas** l'égalité syntaxique. Contre-exemple :

- $P \models Q$ ssi $(P \rightarrow Q) \equiv \top$ ssi $\models P \rightarrow Q$.
- $P \equiv Q$ ssi $(P \leftrightarrow Q) \equiv \top$ ssi $\models P \leftrightarrow Q$.

Preuve :

Proposition 14.

Si $P \equiv Q$ alors pour toute variable $x \in \mathcal{V}$ et toute formule $R \in \mathcal{P}$, on a $P[R/x] \equiv Q[R/x]$.

Démonstration. Soient $P, Q \in \mathcal{P}$ telles que $P \equiv Q$. Soient $R \in \mathcal{P}$ et $x \in \mathcal{V}$. Soit v une valuation quelconque. Il faut montrer que $eval_v(P[R/x]) = eval_v(Q[R/x])$.

Considérons donc la valuation v' définie par :

$$\begin{cases} v'(x) = eval_v(R) \\ v'(y) = v(y) \text{ quand } y \neq x \end{cases}$$

Comme $P \equiv Q$, v' étant une valuation, on a $eval_{v'}(P) = eval_{v'}(Q)$.

Il suffit donc de montrer que $eval_{v'}(P) = eval_v(P[R/x])$ et de même pour Q . On montrera uniquement la propriété pour P , celle pour Q se fera de même. On procède par induction structurale sur $P \in \mathcal{P}$.

- Si $P = \top$ ou \perp , c'est immédiat.
- Si $P = y$ où $y \in \mathcal{V} \setminus \{x\}$: alors $eval_{v'}(P) = eval_v P$ et $P[R/x] = P$, d'où l'égalité.
- Si $P = x$, alors $eval_{v'}(P) = v'(x) = eval_v(R)$. Or, dans ce cas-ci $P[R/x] = R$ et donc $eval_v(P[R/x]) = eval_v(R)$ d'où l'égalité.
- Si $P = P_1 \wedge P_2$, alors :

$$\begin{aligned} eval_{v'}(P) &= Et(eval_{v'}(P_1), eval_{v'}(P_2)) \\ &= Et(eval_v(P_1[R/x]), eval_v(P_2[R/x])) && \text{par H.I.} \\ &= eval_v(P_1[R/x] \wedge P_2[R/x]) && \text{par def de } eval_v \\ &= eval_v(P[R/x]) && \text{par def de la substitution} \end{aligned}$$

- Les cas des autres connecteurs logiques se traitent de même.

La propriété pour Q se fait de même. Comme annoncé, on en déduit que $eval_v(P[R/x]) = eval_v(Q[R/x])$. \square

Proposition 15.

Si $P \equiv P'$ et $Q \equiv Q'$, alors on a les équivalences sémantiques suivantes :

$$P \wedge Q \equiv P' \wedge Q', \quad P \vee Q \equiv P' \vee Q', \quad \neg P \equiv \neg P', \quad P \rightarrow Q \equiv P' \rightarrow Q' \quad \text{et} \quad P \leftrightarrow Q \equiv P' \leftrightarrow Q'.$$

Démonstration. On prouvera uniquement la première équivalence sémantique, les autres se faisant de même.

Soient donc P, Q, P', Q' tels que $P \equiv P'$ et $Q \equiv Q'$ et prouvons $P \wedge Q \equiv P' \wedge Q'$. Soit donc v une valuation. On a :

$$\begin{aligned} eval_v(P' \wedge Q') &= Et(eval_v(P'), eval_v(Q')) \\ &= Et(eval_v(P), eval_v(Q)) \\ &= eval_v(P \wedge Q) \end{aligned}$$

□

Exercice. prouvez les autres cas.

1.2 Satisfiabilité

Définition 16 (Tautologie).

Une **tautologie** est une proposition $P \in \mathcal{P}$ telle que pour toute valuation v , $eval_v(P) = Vrai$. Autrement dit, $\models P$.

Proposition 17.

Une proposition P est une tautologie ssi $P \equiv \top$

Démonstration. Il suffit d'utiliser que pour toute valuation v , $eval_v(\top) = Vrai$.

□

Définition 18 (Satisfiabilité).

Une proposition P est dite **satisfiable** s'il existe une valuation qui la satisfait. On appelle alors une telle valuation un **témoin de satisfiabilité** de P .

Remarque. une formule est donc satisfiable ssi (au moins) une des lignes de sa table de vérité donne Vrai ; et une tautologie ssi toutes ces lignes donnent Vrai.

Définition 19 (Antilogie).

Une **antilogie** (ou contradiction) est une proposition non satisfiable.

Remarque. autrement dit, P est une antilogie ssi il n'existe aucune valuation la satisfaisant ssi toutes les lignes de sa table de vérité donnent Faux.

Proposition 20.

P est une antilogie ssi $P \equiv \perp$

Démonstration. Il suffit d'utiliser que pour toute valuation v , $eval_v(\perp) = Faux$.

□

Proposition 21 (Équivalences sémantiques usuelles).

Soient P, Q, R des propositions. Alors :

- $P \wedge P \equiv P$ (idempotence de \wedge)
- $P \vee P \equiv P$ (idempotence de \vee)
- $\neg\neg P \equiv P$ (double négation)
- $P \vee \neg P \equiv \top$ (tiers exclu)
- $P \wedge \neg P \equiv \perp$ (contradiction)
- $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$ (loi de Morgan)
- $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$ (loi de Morgan)
- $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$ (distributivité de \wedge sur \vee)
- $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$ (distributivité de \vee sur \wedge)

Démonstration. Toutes se font de la même manière. Faisons la première loi de Morgan ensemble : \square

Exercice. : faire le tiers exclu et une des loi de distributivité

Remarque.

- Contrairement à ce à quoi vous êtes peut-être habitués sur les anneaux / espaces vectoriels, ici les deux lois sont distributives l'une sur l'autre.
- **Aucune de ces règles n'est une égalité syntaxique**
- \triangle La double négation et le tiers exclu ne sont valables qu'en "logique classique", dont on explore une petite facette dans ce chapitre (un cas restreint très particulier). Il n'est pas toujours automatiquement vérifié comme ici, et on ne le supposera pas toujours acquis. Dans des cadres logiques plus vaste et plus complets, comme la "logique intuitionniste", on peut très bien par exemple avoir des "propriétés" qui ne sont ni vraies ni fausses. Notamment, la logique intuitionniste interdit le raisonnement "par l'absurde".
- Astuce pratique : vous êtes assez mauvais pour appliquer des distributivités avec les symboles \wedge et \vee . Par contre, vous êtes *excellents* pour appliquer dans \mathbb{R} la distributivité de \times sur $+$. Donc pour appliquer la distributivité que vous voulez, **au brouillon**, remplacez le symbole que vous voulez par \times et l'autre par $+$.

2 Formes normales

2.0 Définitions

Définition 22 (Littéral).

Un **littéral** est une formule composée uniquement d'une variable ou de la négation d'une variable (« x », « $\neg y$ », etc).

Une **clause disjonctive** (resp. **clause conjonctive**) est une disjonction de littéraux (resp. une conjonction de littéraux).

Exemple.

- Voici une clause disjonctive :
- Voici une clause conjonctive :

Remarque. On peut former 2^k clauses disjonctives (resp conjonctives) contenant k littéraux et où chaque variable apparaît une et une seule fois. Pour chaque variable, il faut et il suffit de choisir si elle apparaît positivement ou négatif.

Exemple. avec deux variables a et b :

Définition 23 (Disjunctive / Conjunctive Normal Form).

On appelle :

- **forme normale disjonctive** d'une proposition P (*Disjunctive Normal Form*, ou DNF en anglais) une disjonction de clauses conjonctives logiquement équivalente à P .
- **forme normale conjonctive** d'une proposition P (*Conjunctive Normal Form*, ou CNF en anglais) une conjonction de clauses disjonctives logiquement équivalente à P .

Exemple. Soit $P = (a \vee (c \wedge d)) \wedge (b \vee (c \wedge d))$.

- Une DNF de P est :
- Une CNF de P est :

Remarque.

- Une DNF est « un gros OU (de ET) ».
- Une CNF est « un gros ET (de OU) ».
- On impose parfois⁵ que dans une clause chaque variable apparaisse au plus une fois. En effet, « $x \wedge \neg x$ » n'est pas une clause très pertinente... idem avec un \vee .

5. Et parfois non. Corollaire : bien lire les énoncés !

Convention 24 (Clause vide).

Une disjonction vide est égale à \perp (neutre de \vee pour les évaluations).

$$\bigvee_{x \in \emptyset} x = \perp$$

Une conjonction vide est égale à \top :

$$\bigwedge_{x \in \emptyset} x = \top$$

Ce sont les seules DNF/CNF composées de 0 clauses.

2.1 Construction de DNF/CNF

Théorème 25.

Toute formule propositionnelle admet une DNF et une CNF.

Démonstration. Soit $P \in \mathcal{P}$. Notons x_0, \dots, x_{n-1} ses variables propositionnelles.

- Construisons une DNF pour P .
On trace la table de vérité de P :

Chacune des 2^n lignes correspond exactement à une clause conjonctive contenant chacune des x_i exactement une fois, et donc exactement à une valuation sur les (x_i) . Par exemple, $\neg x_0 \wedge \neg x_1 \wedge \dots \wedge \neg x_{n-1}$ est la première ligne, $x_0 \wedge \neg x_1 \wedge \dots \wedge \neg x_{n-1}$ la seconde, etc etc.

On note C_i la clause correspondante à la i ème ligne. On sélectionne les clauses C_i sur lesquelles P s'évalue à Vrai selon la table :

$$P \equiv \bigvee_{i \text{ tq ligne } i \text{ met } P \text{ vrai}} C_i$$

On a bien trouvé une DNF pour P .

- Construisons maintenant une CNF.

L'idée est de nier une DNF et d'utiliser les lois de Morgan. D'après le point précédent, on sait que $\neg P$ possède une DNF à r clauses possédant chacune n littéraux nommés l_j :

$$\neg P \equiv \bigvee_{i=0}^{r-1} C_i \equiv \bigvee_{i=0}^{r-1} \bigwedge_{j=0}^{n-1} l_j$$

Mais alors, par double négation et lois de Morgan :

$$P \equiv \neg \neg P \equiv \bigwedge_{i=0}^{r-1} \bigvee_{j=0}^{n-1} \neg l_j$$

On a bien trouvé une CNF pour P .



Exemple. Trouver une DNF de la formule suivante :

$$\varphi = (x \wedge \neg z) \rightarrow (\neg x \wedge \neg(y \wedge \neg z))$$

Pour trouver une DNF, traçons la table de vérité de la formule :

x	y	z	$x \wedge \neg z$	$y \wedge \neg z$	$(\neg x \wedge \neg(y \wedge \neg z))$	φ
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

Remarque.

- La lecture du tableau peut aussi permettre d'obtenir une CNF.
- Avec cette façon de construire, chaque clause contient *exactement* une fois chaque variable apparaissant dans P . On parle alors de DNF/CNF canonique et elle est unique à l'ordre des clauses près.
- L'arbre d'une telle DNF/CNF est de hauteur au plus trois... mais peut-être très gros car la racine peut avoir jusqu'à 2^n fils, et ses enfants peuvent avoir n fils (si chaque littéral apparaît au plus une fois).

- La taille d'une DNF/CNF peut-être exponentiellement grande par rapport à la formule initiale.
- Une formule peut avoir une DNF très courte et une CNF très longue ou inversement.

Exemple.

Mettre sous CNF la formule propositionnelle (en DNF) suivante :

$$\varphi = (x_1 \wedge x'_1) \vee (x_2 \wedge x'_2) \vee \dots \vee (x_n \wedge x'_n)$$

2.2 Représentation informatique d'une formule

On a vu comment représenter une formule quelconque en OCaml (sous forme d'un arbre, avec un type somme). On peut faire plus efficace en exploitant le format d'une CNF ou d'une DNF. On adapte nos représentations comme suit :

- **Négation** : On représente la variable x_i par l'entier naturel i , et sa négation (littéral $\neg x_i$) par l'entier négatif $-i$.
- **Clause** : Dans une clause, on sait que les littéraux sont tous séparés par le même symbole \wedge (pour une clause conjonctive) ou \vee (pour une clause disjonctive). Il est donc inutile de les stocker !
On représente une clause $C_i = (l_1 \wedge l_2 \wedge \dots \wedge l_r)$ comme une liste de littéraux $[l_1; l_2; \dots; l_r]$.
- **formule** : Il faut indiquer si la formule est en CNF ou en DNF, et donner chacune de ses clauses (elles sont alors séparées par un même symbole \vee ou \wedge comme précédemment). On a (entre autres) deux possibilités : soit on stocke la formule et son format ensemble, soit on les stocke séparément.

```
1 type literal = int (* i si x_i, -i si not(x_i) *)
2 type clause = literal list
3 type formule = CNF of clause list | DNF of clause list
```



Remarque. On peut aussi vouloir retenir à part le nombre de variables (pour y accéder en $O(1)$), et/ou stocker à part les clauses et l'information DNF/CNF. Dans ce cas, on peut par exemple utiliser un type enregistrement :

```
1 type forme = CNF | DNF
2 type formule = {forme : forme; clauses : clause list; nbvars : int}
```



On pourrait aussi utiliser : `type formule = DNF of (int * clause list) | CNF of ...`

3 SAT et algorithme de Quine

3.0 Problème SAT

Définition 26 (Problème SAT).

On définit le problème de décision SAT :

- **Entrée** : $\varphi \in \mathcal{P}$ une formule propositionnelle.
- **Question** : φ est-elle satisfiable ?

Et ses variantes :

- CNF-SAT idem, mais φ est en CNF.
- k -SAT idem, mais φ est une CNF dont chaque clause contient au plus k littéraux.

Remarque.

- SAT est un problème **central** non seulement en théorie de la complexité mais aussi en pratique. Énormément de problèmes s'encodent assez facilement comme des problèmes SAT ; comme par exemple le Sudoku ou la coloration de graphes (cf TD).
- SAT est un problème a priori « difficile à résoudre », c'est à dire que l'on pense qu'il n'existe pas d'algorithme polynomial pour le résoudre⁶. Rendez-vous en MPI !
- Une autre partie de l'importance de SAT provient du fait qu'un programme n'est jamais qu'une suite d'opérations logiques sur des bits. Étudier SAT revient ainsi à étudier ce qu'un ordinateur peut calculer.

Voici des sous-problèmes classiques de SAT :

- CNF-SAT et les k -SAT sont des restrictions de SAT à des sous-ensembles de formules. Donc si on sait résoudre SAT, on sait automatiquement résoudre CNF-SAT et chacun des k -SAT avec la même complexité que SAT. (On dit que ces problèmes sont *plus faciles* à résoudre que SAT.)
- CNF-SAT est en réalité un problème aussi difficile à résoudre que SAT.
- Si $k \geq 3$, alors k -SAT est un problème aussi difficile que SAT (cf MPI).
- Si $k \leq 2$, on obtient un problème que l'on peut résoudre en temps linéaire (cf TD) à l'aide de théorie des graphes et de tris topologiques !

En revanche :

- DNF-SAT, le problème SAT restreint à des formules sous DNF est trivial à résoudre. En effet, une formule en DNF est satisfiable SSI une (au moins) de ses clauses l'est. On peut donc les traiter indépendamment. Pour chaque clause (conjonctive), on détermine linéairement si elle est satisfiable : il faut et il suffit qu'il existe une valuation qui mette chaque littéral à vrai. (On vérifie qu'on n'a pas un littéral et sa négation dans la même clause en temps linéaire...)

Ceci n'est pas incompatible avec le fait que SAT soit "difficile à résoudre". En effet, chaque formule peut effectivement être mise en DNF, mais cette DNF aura une taille exponentielle par rapport à la formule initiale en général. Cette façon de faire nous donne donc bien une solution exponentielle (en la taille de la formule).

- 1-SAT peut de même trivialement être résolu en temps linéaire (on obtient juste une conjonction de variables).
- 2-SAT peut être résolu en temps linéaire (cf TD et MPI), en se ramenant au problème de calculs de composantes fortement connexes.

6. Et si vous en trouvez un, vous avez prouvé $P \neq NP$ et gagné 1 million de dollars... mais plus probablement vous avez fait une erreur.

Définition 27 (MAX-SAT).

- *Entrée* : une formule propositionnelle φ en CNF.
- *Tâche* : Déterminer le nombre maximal de clauses que l'on peut satisfaire simultanément dans φ .

Remarque. MAX-SAT n'a aucun sens si φ n'est pas en CNF !

3.1 Algorithme de Quine

Remarque. Une solution serait d'explorer exhaustivement les 2^n valuations possibles. Ce n'est pas utilisable en pratique, par exemple pour $n \geq 100$.

En revanche, d'énormes efforts ont été fournis ces dernières décennies pour développer des SAT-solvers capables de résoudre en temps raisonnable des instances "typiques" à plusieurs centaines de milliers de variables. Bien qu'ils restent exponentiels en pire cas, ils ont donc de nombreuses applications pratiques.

On va voir un exemple (basique) de SAT-solver légèrement optimisé : l'algorithme de Quine.

Définition 28 (Arbre de décision).

Un **arbre de décision** est un arbre (parfois même un graphe) dont les arêtes sont étiquetées par des « choix » menant à différents noeuds.

Exemple.

Principe de l'algorithme de Quine : On construit progressivement un arbre de décision marqué par les choix de valeur logique (vrai/faux) donnée à chaque variable. La spécificité de cet algorithme est sa manière d'élaguer cet arbre de décision. Après chaque choix de valeur logique, on simplifie la formule en l'évaluant partiellement, de la façon suivante :

- Si on est ramenés à une formule contenant uniquement \top , on s'arrête : la formule est satisfiable. Notez que cela peut avoir lieu *bien* avant d'avoir donné une valeur à toutes les variables. Ex : $x_0 \vee (x_1 \wedge x_2 \wedge \dots \wedge x_{n-1})$.
- Si on est ramenés à \perp , on s'arrête : la branche en cours de l'arbre n'aboutira jamais à une satisfaction de la formule. L'algorithme va alors essayer d'autres choix.
- On simplifie à chaque étape en appliquant les règles suivantes :

Proposition 29 (Simplif. de Quine).

- $\varphi \wedge \perp \equiv \perp \wedge \varphi \equiv \perp$
- $\varphi \wedge \top \equiv \top \wedge \varphi \equiv \varphi$
- $\varphi \vee \perp \equiv \perp \vee \varphi \equiv \varphi$
- $\varphi \vee \top \equiv \top \vee \varphi \equiv \top$
- $\neg \perp \equiv \top$
- $\neg \top \equiv \perp$
- $\varphi \rightarrow \top \equiv \perp \rightarrow \varphi \equiv \top$
- $\top \rightarrow \varphi \equiv \varphi$
- $\varphi \rightarrow \perp \equiv \neg \varphi$

Exemple. Faire tourner l'algorithme de Quine sur la formule suivante :

$$\varphi = (x \wedge \neg z) \rightarrow (x \wedge \neg(y \wedge \neg z))$$

Tracer bien l'arbre de décision au fur et à mesure à côté!

FIGURE 14.1 – Élagage d'un arbre de décision par l'algorithme de Quine

L'algorithme correspond au pseudo-code suivant :

Algorithme 22 : QUINE

Entrées : φ une formule propositionnelle
Sorties : Vrai si φ est satisfiable ; Faux sinon

```
// Étape essentielle (qui accélère et élague!)
1 Simplifier  $\varphi$ 
2 si  $\varphi = \top$  alors renvoyer Vrai
3 si  $\varphi = \perp$  alors renvoyer Faux

// Sinon : essayer une possibilité, puis si besoin l'autre
4  $x \leftarrow$  une variable présente dans  $\varphi$ 
5 renvoyer QUINE( $\varphi[\perp/x]$ ) || QUINE( $\varphi[\top/x]$ )
```

Remarque.

- Dans le pseudo-code ci-dessus, on suppose disposer du ou logique noté `||` de C/OCaml : il est paresseux, et donc évalue d'abord son opérande gauche et ensuite seulement si nécessaire son opérande droit.
- C'est un algorithme de retour sur trace !
- La notion d'arbre de décision sera revue en MPI, dans le cadre de l'IA.
- Δ Ce pseudo-code seul ne suffit pas, il faut aussi coder la simplification d'une formule⁷, ainsi que la substitution d'une variable par une proposition dans une formule. De plus, il faut un moyen de sélectionner une variable dans φ .

3.2 Une petite parenthèse : les graphe de flots

On peut généraliser la notion d'arbre de décision à celle de graphe de flot (« flowchart »). Il s'agit simplement d'autoriser certains choix à nous faire revenir en arrière.

Un graphe de flot est un graphe orienté étiqueté par les arêtes et les sommets. On essaye en général de se limiter aux arêtes de degré au plus 2 (choix binaires).

Exemple. Les graphes de flot de contrôle sont les graphes de flot représentant l'exécution d'algorithmes en fonction des tests logiques qui y ont lieu.

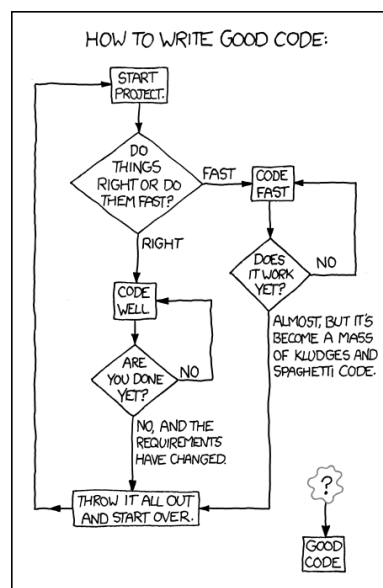


FIGURE 14.2 – Good code. Src : xkcd.com/844

7. C'est l'étape qui contient tout le travail ! cf TP

4 Quantificateurs et introduction à la logique du premier ordre

Jusqu'à présent, nous ne nous sommes intéressé·es qu'au **calcul propositionnel** (avec les formules propositionnelles). Il se trouve que ce cadre est très restreint et ne permet pas de faire grand chose en pratique. En particulier, vous noterez l'absence des quantificateurs \forall et \exists omniprésents dans les théorèmes mathématiques.

Passons maintenant à un formalisme plus général : celui de la logique du premier ordre, que l'on appelle aussi **calcul des prédicats**. On va ajouter deux notions majeures lors de ce changement :

- 1) On ajoute les quantificateurs \forall et \exists à la syntaxe de nos formules
- 2) Les "briques de bases" des formules ne sont plus les variables propositionnelles $x_i \in \mathbb{Z}$, mais n'importe quel «prédicat» (comme par exemple une égalité $x = y$). En particulier, **il n'y a plus aucune raison que les formules construites soient vraies soit fausses**, le tiers exclu et la double négation ne tiendront plus en toute généralité.

Remarque. La logique du premier ordre à proprement parler est hors programme, et on ne verra donc que sa syntaxe, la sémantique étant très explicitement hors programme. Cependant, les quantificateurs ainsi que tout le travail effectué sur les quantificateurs est au programme et à savoir maîtriser, et ce travail perd beaucoup de son sens en dehors du contexte de la logique du premier ordre.

Définition 30.

Un **langage** (du premier ordre) \mathcal{L} est la donnée d'une famille quelconque de symboles divisés en trois sortes :

- les symboles de **fonction**, dotés chacun d'une arité dans \mathbb{N}^* (son nombre d'arguments). Par exemple, une fonction peut être unaire, binaire, etc (n-aire).
- les symboles de **constante**. Il s'agit en quelque sorte de "symboles de fonction d'arité 0".
- les symboles de **relation**, aussi appelées **prédicats**, doté chacun d'une arité $\in \mathbb{N}^*$. On supposera toujours qu'on dispose au moins d'un symbole de relation « = » (égalité) d'arité 2, dans tout langage.

Remarque.

- On appelle aussi un langage \mathcal{L} une **signature**, ou un **vocabulaire**.
- Vous connaissez au moins un autre exemple de symbole de relation classique : \leq ou encore $<$.

Exemple.

Donnons la signature de la théorie des groupes :

Dans toute cette partie, on se dote d'un ensemble infini dénombrable χ de variables. \triangle Ce ne sont plus des variables propositionnelles, mais de simples variables (qui pourront "valoir" n'importe quoi, des entiers, des matrices, etc.).

Définition 31.

L'ensemble \mathcal{T} des termes sur un langage \mathcal{L} est défini inductivement par :

- $\chi \subseteq \mathcal{T}$ (autrement dit, les variables sont des termes).
- les constantes sont des termes : si c est un symbole de constante, alors $c \in \mathcal{T}$.
- Pour tout symbole f de fonction d'arité $n \in \mathbb{N}^*$ et pour tous termes $t_1, t_2, \dots, t_n \in \mathcal{T}$, on a $f(t_1, \dots, t_n) \in \mathcal{T}$

Remarques :

- Attention, nous sommes de nouveau dans le monde de la syntaxe ! $f(t_1, \dots, t_n)$ est une pure écriture de symboles, pas une "application" de fonction, même si c'est comme ça que ce serait **interprété** (dans le monde de la sémantique).
- Autre formulation des termes sur \mathcal{L} : l'ensemble \mathcal{T} est le plus petit ensemble contenant les variables et les constantes et qui est stable par "application" des symboles de fonction de \mathcal{L} .

Exemple.

- Donner quelques termes sur le langage de la théorie des groupes :

De manière équivalente, on peut encore définir les termes comme des arbres. Dessinez l'un des termes précédents (qui n'est pas un cas de base) sous forme d'un arbre :

Définition 32.

Une **formule atomique** sur un langage \mathcal{L} est de la forme $R(t_1, \dots, t_n)$ où R est un symbole de relation n -aire de \mathcal{L} et t_1, \dots, t_n sont des termes sur \mathcal{L} .

Exemple.

Écrire quelques formules atomiques du langage de la théorie des groupes :

Définition 33.

L'ensemble \mathcal{F} des **formules** de la logique du premier ordre est défini inductivement par :

- Les formules atomiques sont des formules.
- Si $\varphi, \varphi' \in \mathcal{F}$, et $x \in \chi$ alors on construit les formules suivantes :
 - $\neg \varphi \in \mathcal{F}$
 - $\varphi \wedge \varphi' \in \mathcal{P}$
 - $\varphi \vee \varphi' \in \mathcal{P}$
 - $\varphi \rightarrow \varphi' \in \mathcal{P}$
 - $\varphi \leftrightarrow \varphi' \in \mathcal{P}$

mais aussi :

- $\exists x, \varphi \in \mathcal{F}$
- $\forall x, \varphi \in \mathcal{F}$

Les symboles \exists et \forall ci-dessus sont appelés des **quantificateurs** :

- \exists (**il existe**) : quantificateur existentiel.
- \forall (**pour tout**) : quantificateur universel.

Remarques :

- Ne pas confondre une *formule* avec une *formule propositionnelle*. L'une est défini dans le cadre de la logique du premier ordre, l'autre dans le cadre du calcul propositionnel. Si vous voulez abréger le second, dites plutôt *proposition* que formule (même si l'abus de langage existe).
- Ces formules sont aussi (de manière équivalente) des arbres. Ex : représenter la formule suivante (du langage de la théorie des groupes) sous forme d'un arbre (où on rappelle que e est un symbole de constante) :

$$\forall x, ((\exists y, x * y = y) \rightarrow x = e)$$

- Les priorités usuelles s'appliquent pour se passer des parenthèses. En particulier, \forall et \exists attrapent tout ce qu'ils peuvent.

Définition 34 (Occurrences libres et liées).

Une **occurrence** d'une variable x dans une formule φ est une position de x dans φ (i.e. un noeud étiqueté par x dans l'arbre associé).

Une occurrence de x est dite **liée** si elle est contenue dans une sous-formule de la forme $\exists x, \varphi'$ ou $\forall x, \varphi'$. Dans ce cas, le dernier quantificateur $\forall x$ ou $\exists x$ rencontré avant x est appelé le **point de liaison** de x .

Si une occurrence de x n'est pas liée, elle est dite **libre**.

Une variable x est dite **libre** si elle possède au moins une occurrence libre, et liée si elle apparaît dans φ et n'est pas libre.

Une formule est dite **close** si elle ne contient aucune variable libre.

On note $V(\varphi)$ l'ensemble des variables qui apparaissent dans φ et $VL(\varphi)$ l'ensemble de ses variables libres.

Définition 35.

La **portée** d'un quantificateur est le sous-arbre enraciné en ce quantificateur, i.e. la portée contient toutes les occurrences de sa sous-formule. Mais Δ , ce n'en est pas forcément le point de liaison.

Exemple.

Remarque.

- C'est comme en programmation, quand on cherche à savoir quelle variable contient quoi et désigne quoi à tout moment du programme. Les notions de portée coïncident.
- Δ Comme en programmation, on évite de donner le même nom de variable à des variables liées par différents points. Ce sont des variables muettes, on peut les renommer.

Définition 36 (Substitution (avec quantificateurs)).

Soit t un terme, φ une formule et x une variable.

La **substitution** de x par t dans φ est la formule obtenue en remplaçant chaque occurrence *libre* de x par dans φ par le terme t .

Remarque.

- Δ Ça n'aurait aucun sens ici de substituer une variable par une formule (en quelque sorte, leurs "types" ne correspondent pas, contrairement au calcul propositionnel où les deux s'évaluent à un booléen Vrai ou Faux).

Et la sémantique alors ? Qu'est ce que ça veut dire tout ça ? C'est Hors-Programme...

Mais pour vous en donner une idée, il ne faut pas seulement interpréter les variables cette fois (comme dans le calcul propositionnel), mais aussi chacun des symboles du langage ! Lorsqu'on a dit ce que "veut dire" chacun de ces symboles, on peut évaluer une formule. Le résultat obtenu dépend fortement de l'**interprétation** (ou **modèle**) qu'on regarde, et n'est plus nécessairement Vrai ou Faux (pensez aux formules indécidables par exemple).

Exemple. Prenons un langage contenant un symbole de fonction $*$ (une loi "multiplication") et un symbole de relation $=$ (égalité). Considérons la formule $\forall x, \forall y, x * y = y * x$ (commutativité de la loi $*$).

Donner une interprétation⁸ du langage tel que cette formule s'évaluerait à Vrai, et une interprétation pour laquelle la formule s'évaluerait à Faux.

8. On ne l'a pas défini formellement, mais vous pouvez déjà le faire intuitivement.