PF1 — Principes de Fonctionnement des machines binaires

Jean-Baptiste Yunès
Jean.Baptiste.Yunes@univ-paris-diderot.fr

Version 1.12

Calcul propositionnel et Algèbre de Boole

Le **calcul propositionnel** ou **calcul des propositions** est une théorie logique qui formalise le raisonnement logique

Exemple : le fameux syllogisme de Socrate :

Tous les hommes sont mortels

or Socrate est un homme

donc Socrate est mortel

2 prémisses qui mènent à 1 conclusion

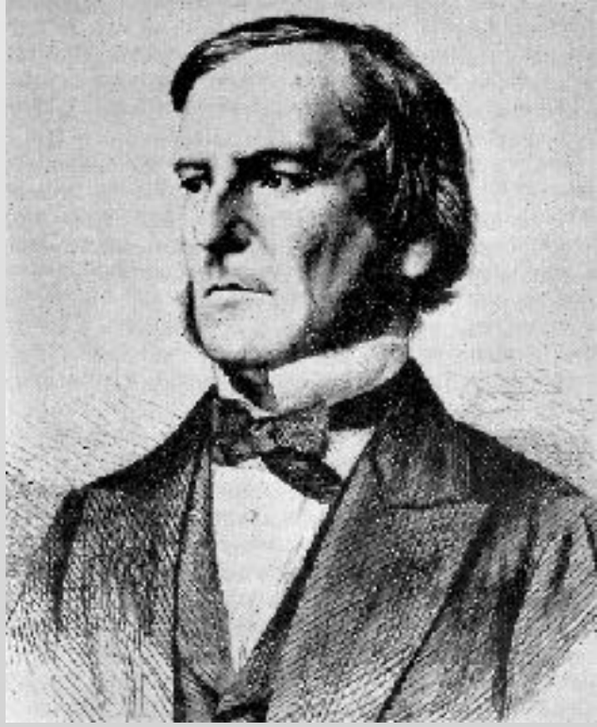
Le calcul propositionnel ne se préoccupe que de la bonne articulation logique des propositions entre elles pas de leur vérité intrinsèque

Un calcul propositionnel **invalide** (paralogisme) :

Tous les imbéciles ont des jambes

or **les informaticiens ont des jambes**

donc **les informaticiens sont des imbéciles**



George Boole
(source Wikipédia)

L'**algèbre de Boole** est le calcul des propositions dont le domaine est celui des booléens **B**, l'ensemble a deux éléments { FAUX, VRAI }, { false, true } ou { 0, 1 } qu'on appelle **valeurs de vérité**…

L'arithmétique s'occupe d'expressions dont le domaine des valeurs est celui des entiers.

Une **proposition** est une entité qui donne une information à propos d'un état des choses

Certaines sont universellement VRAIES ou FAUSSES (ce sont des constantes)

la terre est ronde (VRAI)

les tigres sont carnivores (VRAI)

12 est un nombre premier (FAUX)

Certaines sont parfois VRAIES parfois FAUSSES selon le contexte :

il pleut

nous sommes 2546 dans cet amphi

nous sommes dimanche

Les propositions sont dénommées à l'aide de symboles (**variables propositionnelles** ou **proposition atomiques**), p, q, r, s…

Ce sont les premiers constituants du langage.

on rencontre des propositions un peu partout dans les programmes informatiques

```
x+y < 13
```

cette expression est VRAIE ou FAUSSE selon le contexte

valuation des variables correspondantes

Il existe deux symboles spéciaux destinés à représenter les constantes :

- $\top$ (top) : VRAI
- $\bot$ (bottom) : FAUX

Un langage comme Java possède un type de variables appelé **boolean** avec deux littéraux `true` et `false`

si il est autorisé d'écrire `x+y<13` alors il est autorisé d'écrire:

```
boolean b;
b=x+y<13;
```

En Java les conditionnelles (**`if`**, **`while`**/**`do-while`**, **`for`**, **`?:`**) nécessitent l'emploi d'expression booléenne. Donc toute expression de cette nature peut être employée :

```
if (x<13)…
```

ou

```
boolean b = x>18;
   if (b) …
```

Inutile d'écrire :

```
if (b==true)
```

Les seconds constituants sont les **connecteurs** (ou opérateurs) qui permettent de construire des propositions plus élaborées

un exemple de connecteur est le **ou** qui permet de combiner deux propositions pour en obtenir une plus complexe et dont la valeur de vérité est fonction de la valeur des propositions combinées

p = il pleut

q = il neige

p **ou** q = il pleut **ou** il neige

Les connecteurs n-aires sont donc des applications n-aires de $\mathbf{B}^n \mapsto \mathbf{B}$

Il y a 2 connecteurs 0-aires (les constantes) :

$\top$ et $\bot$

$\bot$	$\top$
$\bot$	$\top$

Il y a 4 connecteurs unaires :

absurdisation, identité, négation, tautologisation

p	absurdisation	négation	identité	tautologisation
⊥	⊥	⊤	⊥	⊤
⊤	⊥	⊥	⊤	⊤

la négation de p est souvent notée ¬p ou $\bar{p}$
En Java elle se note `!`, ex.: `!(x<13)`

Il y a 16 connecteurs binaires :

p	q	$\bot$	et	n-imp	p_1	n-pmi	p_2	x-ou	ou
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$

p	q	n-ou	eq	n-p_2	pmi	n-p_1	imp	n-et	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$

autres notations pour les connecteurs (**Java**, **C**)

et : p and q, p∧q, p.q (**`p&q`**, **`p&&q`**)

ou : p or q, p∨q, p+q (**`p|q`**, **`p||q`**)

x-ou : p xor q, p⊕q (**`p^q`**)

n-et : p nand q, p⊼q

n-ou : p nor q, p⊻q

équivalence : p si et seulement si q, p⇔q, p≡q (**`p==q`**)

implication : si p alors q, p⇒q, p⊃q

Q: combien de connecteurs ternaires existe t-il ?

Le connecteur si-alors-sinon est très connu et employé :

p	q	r	si p alors q sinon r
⊥	⊥	⊥	⊥
⊥	⊥	⊤	⊤
⊥	⊤	⊥	⊥
⊥	⊤	⊤	⊤
⊤	⊥	⊥	⊥
⊤	⊥	⊤	⊥
⊤	⊤	⊥	⊤
⊤	⊤	⊤	⊤

En Java :

condition **?** si-vrai **:** si-faux

Les formules du calcul propositionnel sont donc obtenues par **induction** :

toute constante est une formule

toute variable est une formule

pour tout connecteur unaire *c* et toute formule *f*, *c(f)* est une formule

pour tout connecteur binaire *c* et toutes formules f_1 et f_2, $(f_1)\ c\ (f_2)$ (notation infixe) est une formule

pour tout connecteur n-aire *c* et toutes formules f_i $(0<i\leq n)$, $c(f_1,\ldots,f_n)$ est une formule

Exemple de formules

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

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

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

Certaines parenthèses peuvent être enlevées (pourvu que des **priorités soient définies** entre opérateurs) :

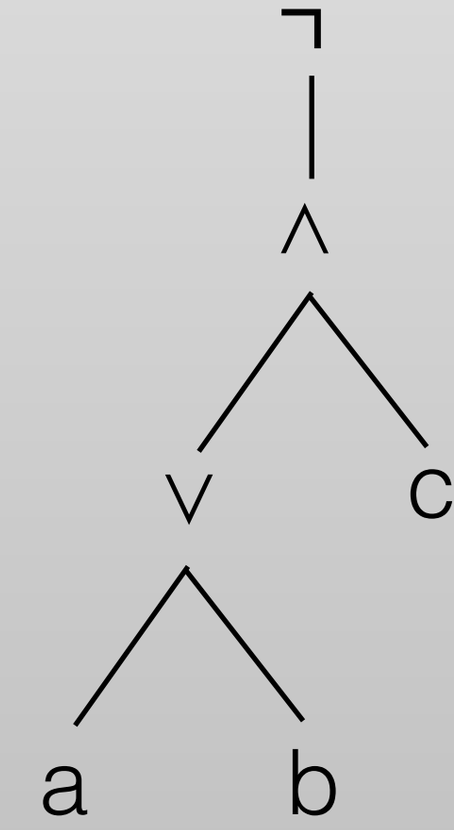
$\neg a \wedge b$

$(a \vee b) \wedge c$

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

Il est parfois pratique d'associer à une formule son **arbre syntaxique** qui reflète sa structure profonde

ainsi pour ¬((a ∨ b) ∧ c) on a



L'arbre syntaxique exprime la formule indépendamment de sa syntaxe formelle

C'est une forme canonique

La forme standard de l'écriture d'une formule (**forme infixe**) a **l'inconvénient** de nécessiter l'usage de parenthèses

il existe deux autres formes que l'on peut extraire facilement de l'arbre syntaxique et qui permettent de s'en passer :

notation polonaise, polonaise inversée.

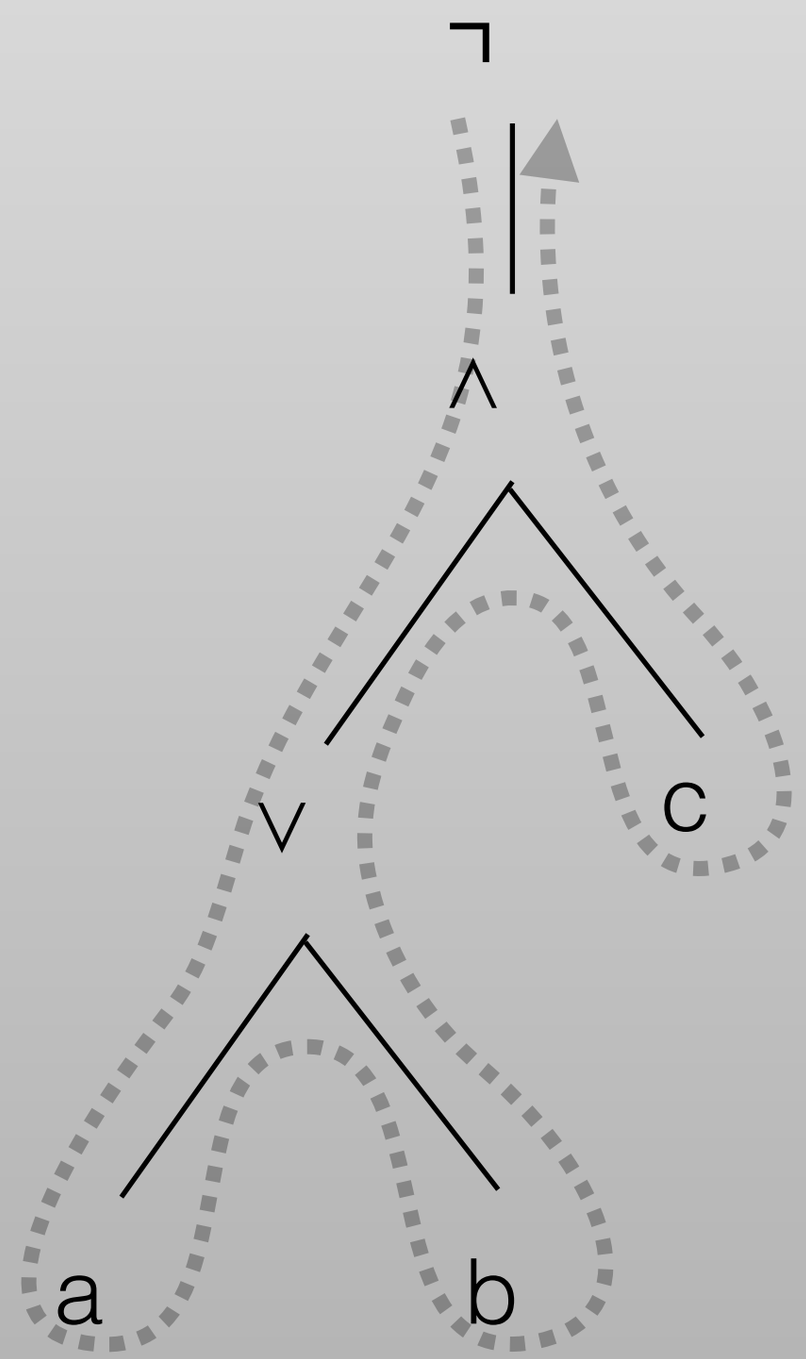
on parle aussi de **forme préfixe ou suffixe** (les opérateurs sont placés après les opérandes ou avant) `+ 1 2` ou `1 2 +` au lieu de `1 + 2`

Jan Łukasiewicz

(source Wikipédia)

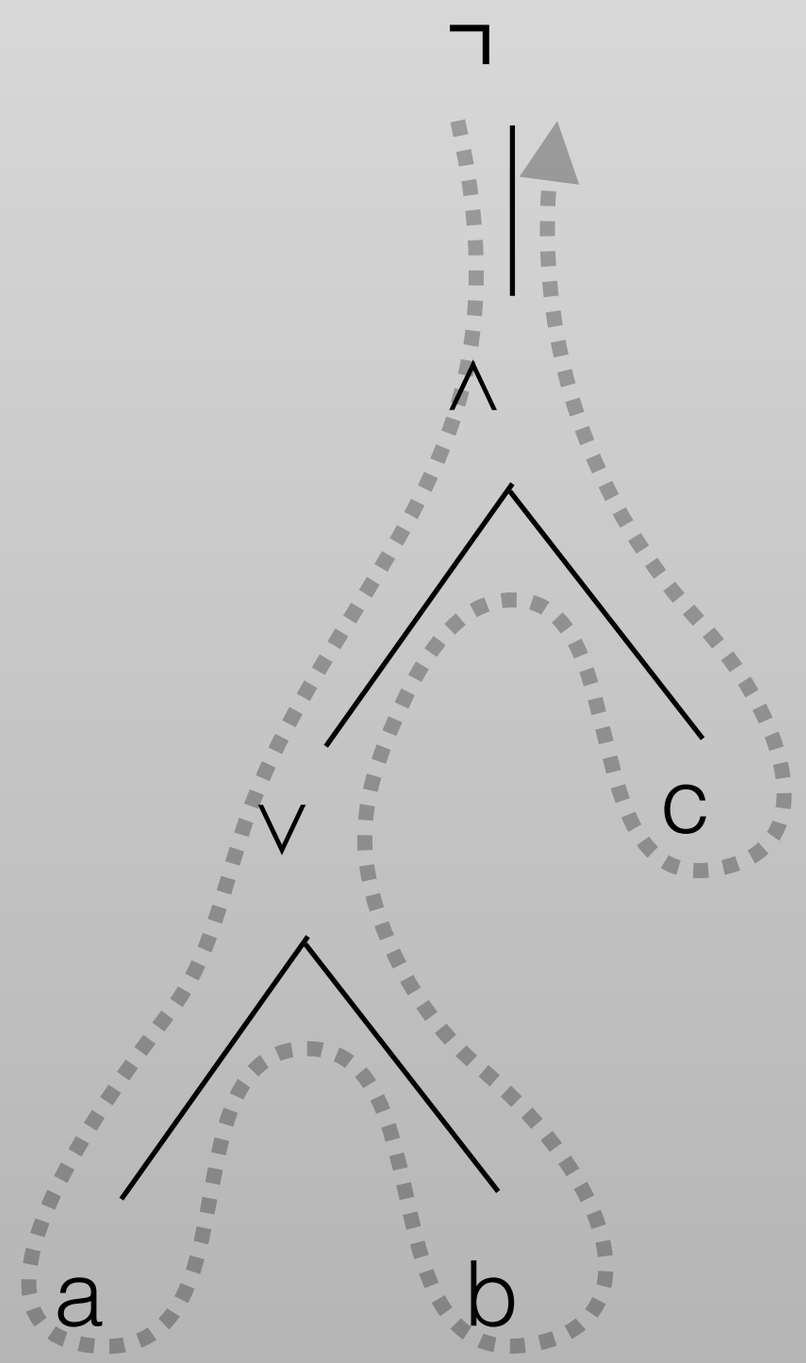
Pour obtenir la **forme préfixe** il faut effectuer un parcours gauche en notant les nœuds à gauche desquels on passe

¬ ∧ ∨ a b c



Pour obtenir la **forme suffixe** il faut effectuer un parcours gauche en notant les nœuds à droite desquels on passe

a b ∨ c ∧ ¬

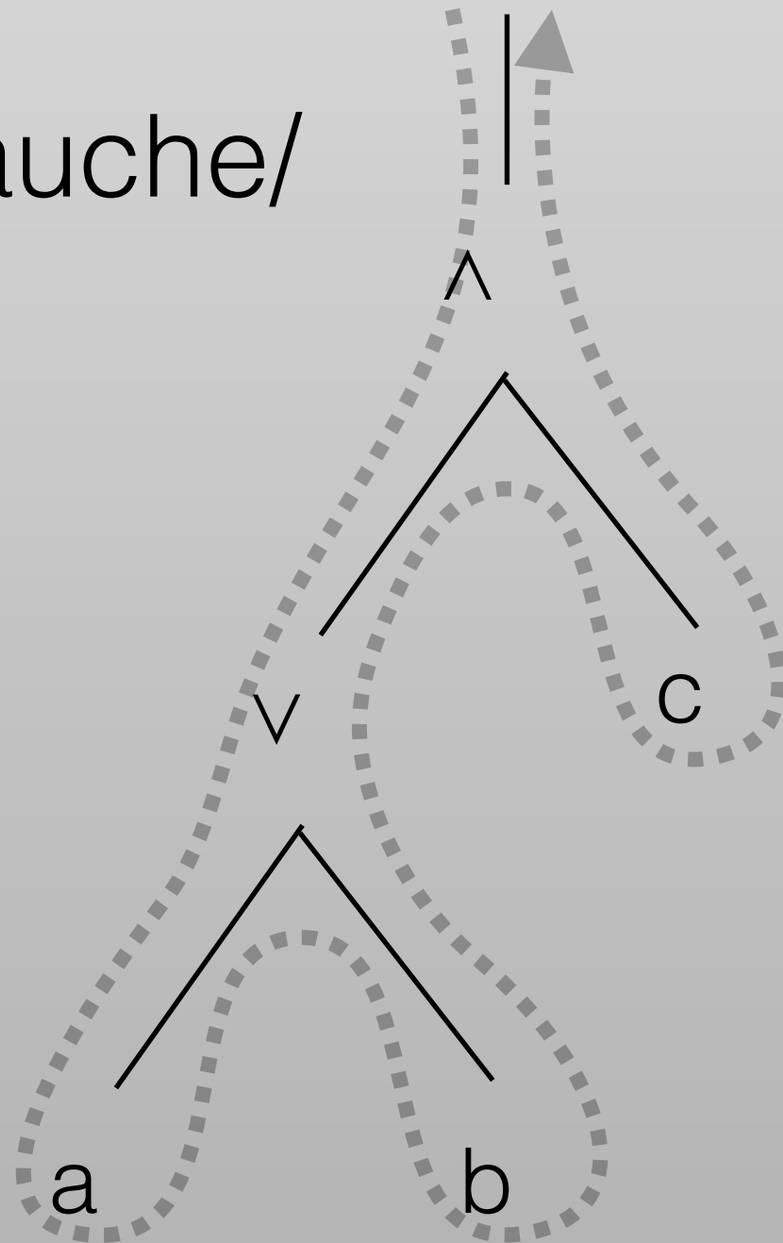


Pour obtenir la **forme infixe** il faut effectuer un parcours gauche en notant les nœuds sous lesquels on passe ¬

Attention il faut parenthéser (gauche/droit)!!!

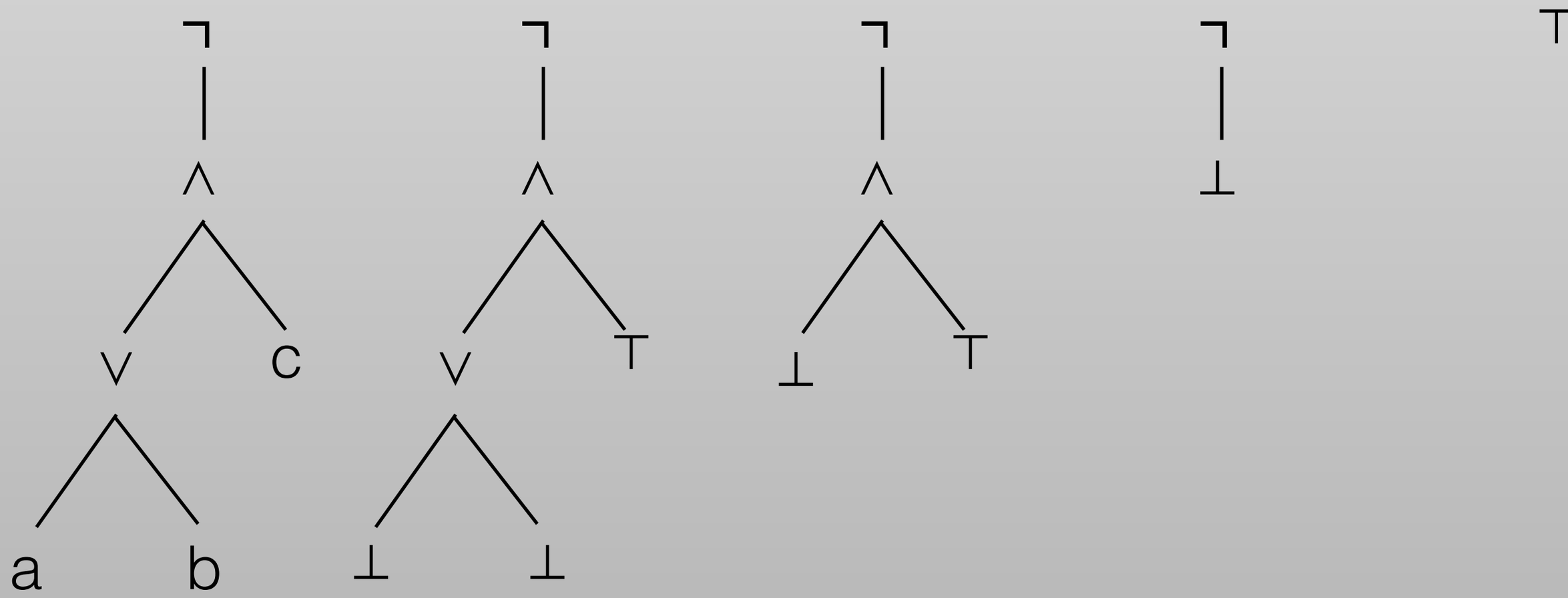
Attention le cas particulier des nœuds unaires

¬(((a)∨(b))∧(c))



L’arbre syntaxique ou arbre de syntaxe abstraite permet aussi d’évaluer une expression étant donnée la valuation des variables

ex : a = ⊥, b = ⊥, c = ⊤



Pour un arbre il existe plusieurs stratégies d’évaluation…

Chaque langage a la sienne…

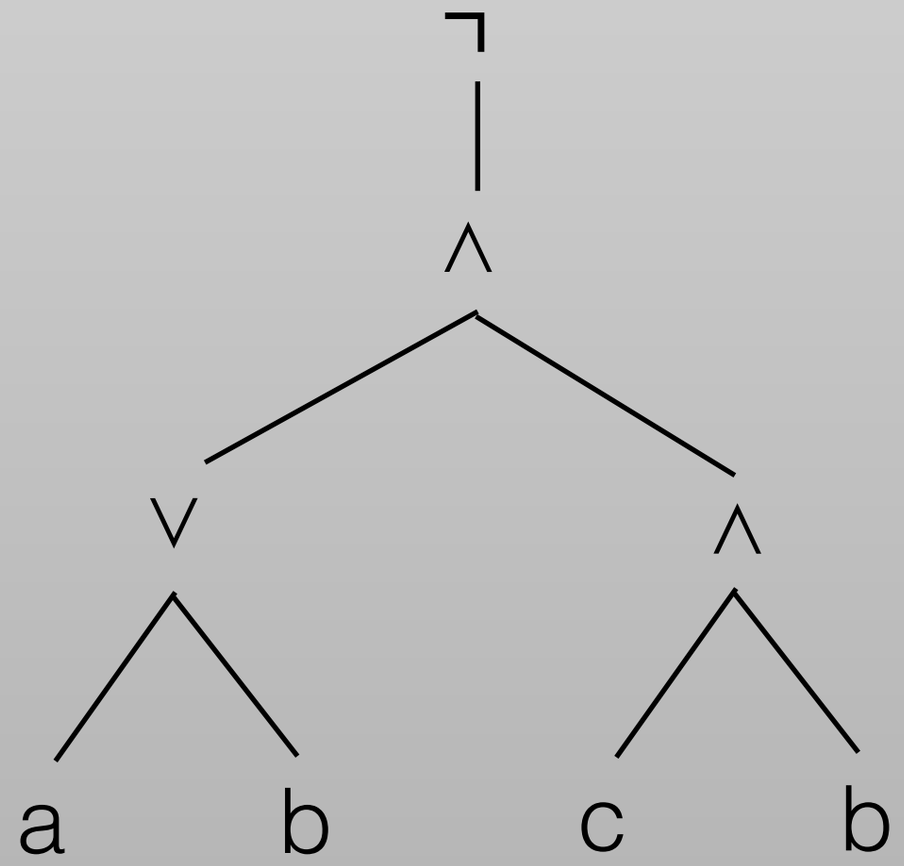


Table de vérité

il est fréquent de présenter les valeurs d'une expression sous la forme d'une table où les entrées sont les valuations des variables de l'expression et les sorties la valuation de l'expression

Q: si une expression possède n variables différentes, combien de valuation différentes sont possibles ? i.e. combien la table aura t-elle de lignes ?

Soit $E_1 = (((a \Rightarrow (\neg b)) \vee (\neg (b \Leftrightarrow c))) \wedge (a \oplus c)$

a	b	c	¬b	a⇒¬b	b⇔c	¬(b⇔c)	(a⇒¬b) ∨ (¬(b⇔c))	a⊕c	E_1
⊥	⊥	⊥	⊤	⊤	⊤	⊥	⊤	⊥	⊥
⊥	⊥	⊤	⊤	⊤	⊥	⊤	⊤	⊤	⊤
⊥	⊤	⊥	⊥	⊤	⊥	⊤	⊤	⊥	⊥
⊥	⊤	⊤	⊥	⊤	⊤	⊥	⊤	⊤	⊤
⊤	⊥	⊥	⊤	⊤	⊤	⊥	⊤	⊤	⊤
⊤	⊥	⊤	⊤	⊤	⊥	⊤	⊤	⊥	⊥
⊤	⊤	⊥	⊥	⊥	⊥	⊤	⊤	⊤	⊤
⊤	⊤	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥

Quelques propriétés des opérateurs logiques

la négation est **involutive**, c'est-à-dire que ¬¬a ⇔ a

les connecteurs ∧ et ∨ sont **idempotents**, c'est-à-dire que a∨a ⇔ a et a∧a ⇔ a

simplifications fondamentales

a∨⊥ ⇔ a, a∨⊤ ⇔ ⊤, a∨(¬a) ⇔ ⊤

a∧⊥ ⇔ ⊥, a∧⊤ ⇔ a, a∧(¬a) ⇔ ⊥

commutativité, associativité

Quelques propriétés des opérateurs logiques

absorption, $(a \wedge b) \vee a \Leftrightarrow a$, $(a \vee b) \wedge a \Leftrightarrow a$

distributivité

de $\vee$ par rapport à $\wedge$: $a \vee (b \wedge c) \Leftrightarrow (a \vee b) \wedge (a \vee c)$,
$a+(b.c) \Leftrightarrow (a+b).(a+c)$

de $\wedge$ par rapport à $\vee$: $a \wedge (b \vee c) \Leftrightarrow (a \wedge b) \vee (a \wedge c)$,
$a.(b+c) \Leftrightarrow (a.b) + (a.c)$

loi de De Morgan

$\neg(a \vee b) \Leftrightarrow \neg a \wedge \neg b$
$\neg(a \wedge b) \Leftrightarrow \neg a \vee \neg b$

Auguste De Morgan
1806-1871
(source Wikipedia)

On se focalise généralement sur les opérateurs $\vee$, $\wedge$ et $\neg$, pourquoi ?

toute formule logique peut s'écrire en utilisant uniquement ces opérateurs…

un ensemble d'opérateurs est complet si toute formule logique peut s'écrire en utilisant uniquement ses opérateurs

$\{\vee, \wedge, \neg\}$ est complet

$\{\wedge, \neg\}$ est complet

Il y d'autres propriétés des systèmes logiques intéressantes à considérer :

peut-on déterminer si une proposition est un théorème ?

la consistance/cohérence : existe-t-il une proposition à la fois vraie et fausse ?

Pour montrer que {∨,∧,¬} est complet, il suffit de montrer que n'importe quelle formule a une forme équivalente n'utilisant que ces trois connecteurs

Toute formule a une **forme normale disjonctive unique** qui est une expression de la forme $\bigvee_{i=1}^{i=m} (\bigwedge_{j=1}^{j=n} x_j)$

où m<n, n est le nombre de variables de la formule et x_j les variables ou leurs compléments (i.e. v_i ou $\neg v_i$)…

cette écriture est une **disjonction de conjonction** (une somme de produits)

les $\bigwedge_{j=1}^{j=n} x_j$ sont appelés des **mintermes**

Une conjonction logique de deux évènements représente leur simultanéité, les évènements sont conjoints

Une disjonction logique de deux évènements représente l'affirmation qu'au moins un des deux évènements est vrai (attention ce n'est pas la disjonction du langage courant)

Il existe aussi une **forme normale conjonctive**

une conjonction de disjonction

des et de ou (un produit de sommes)

Construction de l'expression sous FND

dans la table de vérité de l'expression considérer, ne retenir que les lignes pour lesquelles la formule a pour valeur ⊤

pour chaque telle ligne, on fabrique un **minterme** dans lequel on fait apparaître v_i si sa valeur est ⊤, et $\neg v_i$ sinon

par exemple pour

a	b	c	E
⊥	⊤	⊥	⊤

on a ¬a.b.¬c

pour finir, on fait la disjonction des tous les mintermes obtenus

$E_1 = (((a \Rightarrow (\neg b)) \vee (\neg (b \Leftrightarrow c))) \wedge (a \oplus c)$

Quelle est son écriture en FND ?

a	b	c	E_1
⊥	⊥	⊥	⊥
⊥	⊥	⊤	⊤
⊥	⊤	⊥	⊥
⊥	⊤	⊤	⊤
⊤	⊥	⊥	⊤
⊤	⊥	⊤	⊥
⊤	⊤	⊥	⊤
⊤	⊤	⊤	⊥

Un langage comme Java permet d’exprimer des formules du calcul propositionnel mais aussi de manipuler les nombres vus comme des suites de bits représentant des valeurs de vérité à l’aide de connecteurs

Expressions booléennes en Java

le langage possède un type de données pour représenter les booléens : **`boolean`**, on peut les utiliser avec les connecteurs suivants (entre autres) :

`&&` est l'opérateur booléen fainéant de conjonction, `&` est l'opérateur par évaluation stricte

`||` est l'opérateur booléen de disjonction, `|` est l'opérateur par évaluation stricte

`!` est l'opérateur de négation

on peut donc écrire des choses comme

```
if (a && (!b|c)) {} else {}
```

Indiquons d’abord qu’une expression est habituellement évaluée de gauche à droite (sens de lecture) 1+2+3 c’est (1+2)+3

Qu’est-ce que l’évaluation fainéante ? stricte ?

L’évaluation fainéante est une optimisation permettant d’éviter de considérer la suite de l’expression si la valeur finale est déjà connue…

que penser de $a \vee b$, si $a \Leftrightarrow \top$???

Les opérateurs bits-à-bits ne s'utilisent (en Java) que sur des types entiers (`byte, char`, `short`, `int, long`), on a

&, la conjonction bit-à-bit

|, la disjonction bit-à-bit

^, le ou-exclusif bit-à-bit

~, la négation bit-à-bit

les décalages (déjà vus)…

Retour sur l'addition et la multiplication binaire

Prenons la table d'addition des chiffres de la base 2

+	0	1
0	0	1
1	1	(1)0

C'est un connecteur d'arité 2 (si on oublie la retenue)

Il s'agit même du $\oplus$ (xor)

Pour la table de multiplication on a

.	0	1
0	0	0
1	0	1

C’est aussi un connecteur d’arité 2

Il s’agit du $\wedge$ (et)

Le problème de la retenue pour l'addition…

Il suffit de voir l'addition binaire comme deux fonctions d'arité 3. L'une calculant le résultat et l'autre la retenue (suivante) en utilisant les deux bits et une retenue (précédente)

a	b	r	+
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

a	b	r	R
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1