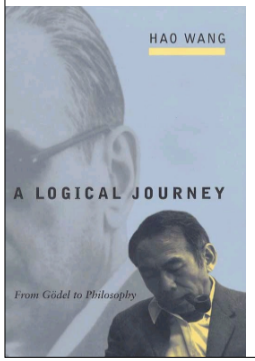
Université Nice Sophia Antipolis

Logique et Démonstation Automatique



La Logique et l'I.A.

• En 1956 : le **Dartmouth Summer Research Project on "Artificial Intelligence"**. Le terme IA est alors proposé par **John McCarthy**, l'inventeur de Lisp [à l'origine de Scheme] au MIT.

• L'IA se propose en particulier de modéliser le raisonnement humain, au moins sous sa forme logique.

SI il fait beau
ALORS je vais à la plage — $p \Rightarrow q$

SI il fait beau
ALORS je vais à la plage OU à la montagne — $p \Rightarrow q \vee r$

SI il fait beau ET je suis en forme
ALORS je vais à la plage OU à la montagne — $p \wedge q \Rightarrow r \vee s$

SI il ne dort pas
ALORS il mange — $\sim p \Rightarrow q$

• Les connecteurs logiques usuels :

$\vee$	$\wedge$	$\neg$	$\Rightarrow$
ou	*et*	*non*	*implique*

• Et leurs tables de vérité :

p	true	false
¬p	false	true

	p	true	true	false	false
	q	true	false	true	false
conjonction	p ∧ q	true	false	false	false
disjonction	p ∨ q	true	true	true	false
implication	p ⇒ q	true	false	true	true

• John McCarthy fut le premier à remarquer que dans un langage de programmation [Lisp], **les opérateurs logiques ne sont pas des fonctions** et que déjà dans $p \wedge q$, la valeur de p peut suffire à obtenir la valeur de l'expression [par ex. $Faux \wedge q \equiv Faux$ pour tout q]. Idem en Java/C avec la construction p && q.

• Ces 4 connecteurs ne sont pas indépendants. Par exemple, la loi de De Morgan établit que la formule $p \wedge q$ pourrait être définie par :

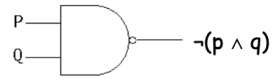
$$p \wedge q \;:=\; \neg(\neg p \vee \neg q)$$

L'implication

• De même la formule $p \Rightarrow q$ est *définie* par $\neg p \vee q$: ***une implication est toujours vraie sauf si l'hypothèse est vraie et la conclusion est fausse***. La phrase suivante est donc vraie :

Si le Nil est en Russie alors Paris est la capitale de la France.

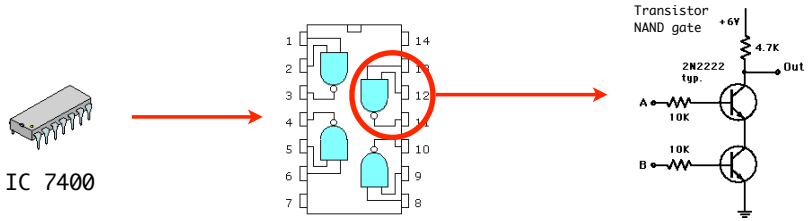
• Les électroniciens utilisent dans leurs puces l'opérateur NAND défini par :



(nand p q) ≡ (not (and p q))

• On peut montrer que cet opérateur NAND engendre à lui tout seul tous les autres !

• Problème d'architecture des ordinateurs : minimiser le nombre de portes NAND pour réaliser un circuit logique donné...



• La Logique d'ordre 0 [ou Logique des Propositions] porte sur des propositions constantes p, q...

$$p \wedge (\neg q \vee r)$$

• La Logique d'ordre 1 [ou Logique des Prédicats] porte sur des propositions utilisant des variables p(x), q(x,y,z), etc. Par exemple P est vu comme un prédicat à une variable :

$$p(x) \wedge (\neg q(x,y) \vee r(y))$$

• Les variables peuvent être quantifiées :

$$\forall x\ \exists y : p(x) \wedge (\neg q(x,y) \vee r(y))$$

• La Logique d'ordre 2 traiterait les prédicats eux-mêmes comme des variables. En Scheme, c'est la *programmation d'ordre supérieur* pour laquelle les fonctions sont des variables !

• Dans ce qui suit, le cadre d'étude sera celui de la **logique d'ordre 0**.

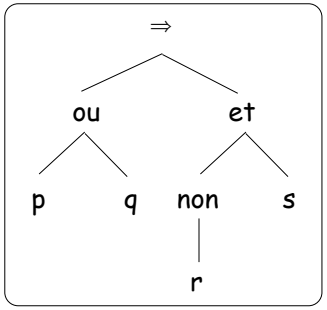
Une formule logique est un ARBRE

GRAMMAIRE DES FORMULES

• Une **formule bien formée** est un ***atome*** ou une ***molécule***.

• Un **atome** sera un symbole p, q, r...

• Une **molécule** portera un opérateur et :
- **un seul fils** [noeud unaire *non*]
- ou bien **deux fils** [noeuds binaires *ou et* ⇒]



p ∨ q ⇒ ¬r ∧ s

N.B. Les fils sont aussi des formules ! — *en fait des arbres 1-2*

```
fbf       ::= ATOME | MOLECULE
ATOME     ::= p | q | r | ...
MOLECULE  ::= (fbf et fbf) | (fbf ou fbf) | (fbf => fbf) | (non fbf)
```

```
(define (atome? x)              ; le reconnaisseur d'atomes [symboles p, q, r...]
  (symbol? x))

(define (make-fbf1 r Ag)        ; le constructeur de molécule à 1 seul fils (négation)
  (if (equal? r 'non) (list r Ag) (error 'make-fbf1 "Mauvaise formule !")))

(define (make-fbf2 r Ag Ad)     ; le constructeur de molécule binaire (et, ou, =>)
  (if (not (member r '(et ou =>)))
    (error 'make-fbf2 "Mauvais connecteur" r)
    (list Ag r Ad)))            ; représentation interne infixée

(define (connecteur mol)        ; on suppose que mol est une molécule
  (if (= (length mol) 2)
    (first mol)                 ; non
    (second mol)))              ; et, ou, =>

(define (arg1 mol)              ; mol est une molécule
  (if (= (length mol) 2)
    (second mol)
    (first mol)))

(define (arg2 mol)              ; mol est une molécule
  (if (= (length mol) 2)
    (error 'arg2 "Molécule unaire" mol)
    (third mol)))
```

Type abstrait "formule bien formée"

• Il s'agit maintenant d'utiliser ce *type abstrait* pour analyser ou construire des fbf, bref programmer avec des arbres logiques.

~~UNE FORMULE EST UNE LISTE~~

UNE FORMULE EST UN ARBRE

• Pour les tests uniquement, il y a deux manières de définir un arbre :

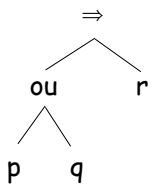
- soit en ***syntaxe abstraite*** :

```
(define AT (make-fbf2 '=> (make-fbf2 'ou 'p 'q) 'r))
```

- soit en ***syntaxe concrète*** en violant le type abstrait :

```
(define AT '((p ou q) => r))
```

car je sais qu'une formule est une liste !



• Nous avons vu au cours 10 des algorithmes sur les arbres.

L'algorithme de Wang

• Il s'agit d'un algorithme du Calcul des Propositions [donc en Logique d'ordre 0] permettant de **décider si une formule est un théorème**. Donc en quelque sorte un ***démonstrateur automatique*** [*prover*]...

• Une **formule bien formée** [fbf] fait intervenir :
- des formules atomiques p, q, r, ... et deux constantes `true` et `false`
- des connecteurs binaires **et**, **ou**, =>
- le connecteur unaire **non**

• La syntaxe concrète est analogue à celle de Scheme mais infixée :

```
fbf      ::= ATOME | MOLECULE
ATOME    ::= p | q | r | ...
MOLECULE ::= (fbf et fbf) | (fbf ou fbf) | (fbf => fbf) | (non fbf)
```

• Exemple : p ∧ (q => r) => p ∨ ¬q sera représentée par :

```
((p et (q => r)) => (p ou (non q)))
```

• Une formule est un **théorème** si elle est toujours vérifiée, quelles que soient les valeurs des variables. Exemples de théorèmes :

```
(p ou (non p))
((p et (p => q)) => q)
```

• Mais la formule `(p => (p et q))` n'est pas un théorème, elle est fausse si par exemple `p = true` et `q = false` !

• **L'algorithme de Wang** permet de **décider si une formule est un théorème, sans utiliser de table de vérité** !

• Il consiste à appliquer une série de **règles** jusqu'à réduction à une forme irréductible où l'on peut décider à vue.

Toward Mechanical Mathematics
Hao Wang
IBM Journal, January 1960
(IBM 704 : 220 theorems in 3 minutes...)

• La forme générale d'un théorème est :

si HYPOTHESES alors CONCLUSION

• Wang utilise le ***calcul des séquents***. Un **séquent** est représenté symboliquement par :

$$H_1 \wedge \ldots \wedge H_n \vdash C_1 \vee \ldots \vee C_k$$

qui se lit intuitivement : *"Si toutes les hypothèses H_1 ,..., H_n sont vérifiés, alors au moins l'une des conclusions C_1 ,..., C_k est vérifiée".*

• Le calcul des séquents permet de modifier l'ordre des formules dans un séquent sans changer sa validité. Par exemple, on peut changer de côté une formule atomique en prenant sa négation :

équivaut à :

$$(p) \wedge (p \Rightarrow q) \vdash q$$

$$p \Rightarrow q \vdash (\neg p) \vee q$$

d'où la validité puisque $p \Rightarrow q$ est défini comme valant $\neg p$ *ou* q

• Règle R1 : une négation peut être changée de côté en enlevant le connecteur ¬ :

$$\zeta \vdash \phi, \neg\lambda, \rho \quad \rightsquigarrow \quad \zeta, \lambda \vdash \phi, \rho$$

• Règle R2 : dissolution d'un ou sur la droite, et d'un et sur la gauche.

$$\phi \vdash \lambda, \rho \vee \pi \quad \rightsquigarrow \quad \phi \vdash \lambda, \rho, \pi$$
$$\phi, \zeta \wedge \rho \vdash \lambda \quad \rightsquigarrow \quad \phi, \zeta, \rho \vdash \lambda$$

• Règle R3 : dédoublement des lignes lors d'un ou sur la gauche.

$$\phi, \zeta \vee \rho \vdash \lambda \quad \rightsquigarrow \quad \begin{cases} \phi, \zeta \vdash \lambda \\ \phi, \rho \vdash \lambda \end{cases}$$

• Règle R4 : dédoublement des lignes lors d'un et sur la droite.

$$\phi \vdash \lambda, \zeta \wedge \rho \quad \rightsquigarrow \quad \begin{cases} \phi \vdash \lambda, \zeta \\ \phi \vdash \lambda, \rho \end{cases}$$

• Un exemple de formule qui n'est <u>pas</u> un théorème : $(p \Rightarrow q) \Rightarrow p$

$\vdash (p \Rightarrow q) \Rightarrow p$

$\vdash \neg(\neg p \vee q) \vee p$ *(élimination des ⇒)*

$\vdash \neg(\neg p \vee q)\ ,\ p$ (R2)

$\neg p \vee q \vdash p$ (R1)

$$\begin{cases} \neg p \vdash p \\ q \vdash p \end{cases}$$ (R3)

$$\begin{cases} \vdash p, p \\ q \vdash p \end{cases}$$ Je suis coincé, INVALIDE !

• Nous allons programmer une fonction `(valide? fbf)` prenant une formule `fbf` et retournant `true` si c'est un théorème...

• Règle R5 : un séquent $\lambda \vdash \zeta$ est un **théorème** si une même formule apparaît de chaque côté de la flèche !

• Le jeu consiste donc à éliminer peu à peu toutes les molécules jusqu'à la preuve par R5, ou l'échec si plus rien ne s'applique !

• Exemple. On cherche à prouver la formule $[p \wedge (p \Rightarrow q)] \Rightarrow q$

On considère l'embryon de séquent : $\varnothing \vdash [p \wedge (p \Rightarrow q)] \Rightarrow q$

dont la partie gauche est vide, et on va le travailler pour converger sur une forme décidable. **On commence par éliminer les implications :**

$\vdash \neg[p \wedge (\neg p \vee q)] \vee q$

On applique les règles :

$\vdash \neg[p \wedge (\neg p \vee q)]\ ,\ q$ (R2)

$p \wedge (\neg p \vee q) \vdash q$ (R1)

$p\ ,\ \neg p \vee q \vdash q$ (R2)

(R3) $\begin{cases} p\ ,\ \neg p \vdash q \\ p\ ,\ q \vdash q \end{cases}$ devient $p \vdash p\ ,\ q$ (R1)

VALIDES par (R5)

CQFD !...

Une implémentation de l'algorithme de Wang

VIII. A COMPLETE LISP PROGRAM - THE WANG ALGORITHM FOR THE PROPOSITIONAL CALCULUS

(LISP 1.5 Programmer's Manuel, MIT PRESS, 1958, p.44)

• Tout naturellement avec des **listes**.

• Reconnaissance des **formules bien formées** [fbf?] :

```
> (fbf? '((a et (non b)) => a))
#t
> (fbf? '(a => non b))
#f
```

```
> (fbf? 'p)
#t
> (fbf? '())
#f
```

• Laissons-nous guider par la grammaire des fbf [page 6] :

```
fbf       ::= ATOME | MOLECULE
ATOME     ::= p | q | r | ...
MOLECULE  ::= (fbf et fbf) | (fbf ou fbf) | (fbf => fbf) | (non fbf)
```

• Prélude : élimination des implications dans une formule. On travaille en profondeur pour rechercher les symboles ⇒

```
(define (reformat fbf)
  (if (atome? fbf)
      fbf
      (local [(define r (connecteur fbf))]
        (cond ((equal? r 'non) ...)
              ((member r '(et ou)) ...)
              (else ...)))))
```

```
> (reformat '((p => q) => p))
((non ((non p) ou q)) ou p)
```

cf TP !

• But : la règle R5. La même formule apparaît-elle de chaque côté de la flèche d'un séquent ? L'intersection de deux listes est-elle non vide ?

```
(define (intersection? L1 L2)          ; au moins un élément en commun ?
  (cond ...))
```

cf TP !

```
> (intersection? '(a b c d e) '(g h k m))
#f
> (intersection? '(a b c d e) '(g h b k))
#t
```

• La fonction (théorème? fbf) se contente d'initialiser un séquent sans implications dont la partie gauche [*lhs*] est vide, et passe la main au moteur de Wang :

```
(define (théorème? fbf)
  (if (not (fbf? fbf))
      (error ...)
      (wang ...)))
```

cf TP !

```
> (théorème? 'p)
false
> (théorème? '(p => p))
true
> (théorème? '(p => (p ou q)))
true
> (théorème? '((p et (p => q)) => (p et q)))
true
> (théorème? '(p => (p et q)))
false
```

• Le moteur de Wang. Il va essayer de ***saturer sur les règles*** jusqu'à ce que (R5) s'applique ou que plus rien ne s'applique !

```
(define (wang LHS RHS)            ; LHS et RHS sont des listes de fbf sans implications
  (if (intersection? LHS RHS)
      true ←————————————— GAGNE !
      (local [(define fbf (first-molecule LHS))]    ; on cherche une molécule à gauche
        (if (not (equal? fbf false))
            (local [(define r (connecteur fbf))]
              (cond ((equal? r 'non) (wang ...)) ←——— négation à gauche
                    ((equal? r 'ou) (and (wang ...) (wang ...))) ←— disjonction à gauche
                    ((equal? r 'et) (wang ...)) ←——— conjonction à gauche
                    (else (error "Syntaxe incorrecte" fbf))))
            (local [(define fbf (first-molecule RHS))]      ; sinon on cherche à droite
              (if (not (equal? fbf false))
                  (local [(define r (connecteur fbf))]
                    (cond ((equal? r 'non) (wang ...))
                          ((equal? r 'ou) (wang ...))
                          ((equal? r 'et) (and (wang ...) (wang ...)))
                          (else (error 'wang "Syntaxe incorrecte"))))
                  false))))))  ←————————————————— ECHEC
```

cf TP !

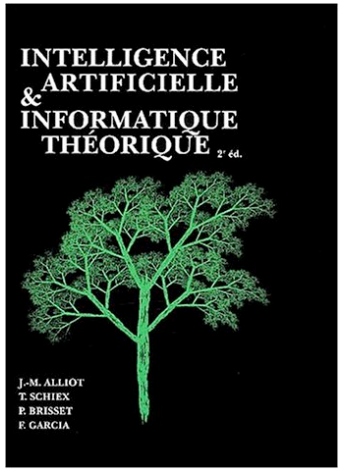
Lisp et l'I.A.

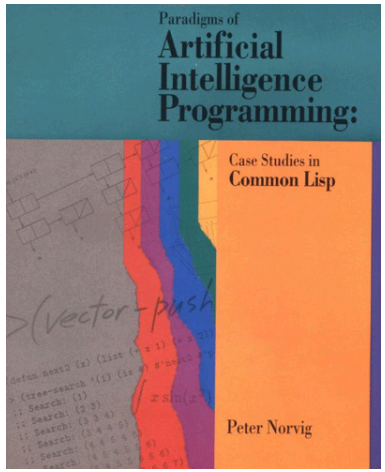
• Le langage **LISP** [LISt Processor] est né vers 1958 pour :
- les besoins de l'**Intelligence Artificielle** naissante [1956]
- disposer d'un modèle exécutable du **λ-calcul** [1940]

• L'I.A. privilégie les problèmes ***symboliques*** sur les problèmes ***numériques***. Manipuler des concepts plutôt que des nombres !

• L'idée de ***mécaniser la Logique*** remonte à Leibniz et à Boole. Mais c'est avec les premiers logiciels d'I.A. que sont apparus des démonstrateurs de théorèmes, comme le "Logic Theorist" de Newell & Simon en 1956. L'algorithme de Wang en Logique des Propositions [logique d'ordre 0] est décrit dans le manuel cité de LISP 1.5 [1958].

• La ***démonstration automatique*** fut ensuite étendue à la Logique des Prédicats [l'ordre 1] où les propositions contiennent des variables : création du langage **PROLOG** à Marseille en 1972.





https://fr.wikipedia.org/wiki/Intelligence_artificielle