# Preuves assistées par ordinateur

## CM 2 - Quelques compléments sur LEAN

4 avril 2025

# Objectif de la séance

Au cours de cette séance, nous allons présenter quelques points supplémentaires sur le fonctionnement de LEAN.

Tout cela ne nous servira pas directement dans l'écriture des preuves, mais cela pourra nous donner une meilleure idée de la façon dont le logiciel fonctionne.

# La "curryfication" : $(P$ et $Q) \Rightarrow R$ vs. $P \Rightarrow (Q \Rightarrow R)$.

## Exercice

Si $P, Q, R$ sont des propositions, alors les deux propositions

$$P \Rightarrow (Q \Rightarrow R) \quad \text{et} \quad (P \text{ et } Q) \Rightarrow R$$

ont la même table de vérité.

| $P$ | $Q$ | $R$ | $P$ et $Q$ | $Q \Rightarrow R$ | $P \Rightarrow (Q \Rightarrow R)$ | $(P$ et $Q) \Rightarrow R$ |
|---|---|---|---|---|---|---|
| V | V | V | V | V | V | V |
| V | V | F | V | F | F | F |
| V | F | V | F | V | V | V |
| V | F | F | F | V | V | V |
| F | V | V | F | V | V | V |
| F | V | F | F | F | V | V |
| F | F | V | F | V | V | V |
| F | F | F | F | V | V | V |

$$P \Rightarrow (Q \Rightarrow R) \qquad \Leftrightarrow \qquad (P \text{ et } Q) \Rightarrow R$$

Autrement dit, les phrases

- « **Si** $P$ est vraie, **alors si** $Q$ est vraie, **alors** $R$ est vraie. »
- « **Si** $P$ et $Q$ sont vraies, **alors** $R$ est vraie. »

sont équivalentes.

Les deux énoncés suivants aboutissent en fait à démontrer le même résultat :

```
- Dans la suite, P Q R sont des propositions
theorem exemple_v1 : P→(Q → R) := by
{
    - Arguments de la preuve
}
theorem exemple_v2 : (P ∧ Q) → R := by
{
    - Arguments de la preuve
}
```

Dans les TP, on a peu utilisé la première version, mais elle est en fait un peu plus pratique d'utilisation.

```
theorem exemple_1 :
  ∀ x y : Int, (x = 3 ∧ y = 2 ) → (x + y = 5) := by
{
  intro  x y
  intro H
  rw [H.left, H.right]
  simp
}
theorem exemple_2 :
  ∀ x y : Int, (x = 3)  →  (y = 2  →  x + y = 5) := by
{
  intro  x y
  intro H1
  intro H2
  rw [H1, H2]
  simp
}
```

Notez que les écritures

$$\texttt{P} \to (\texttt{Q} \to \texttt{R}) \quad \text{et} \quad \texttt{P} \to \texttt{Q} \to \texttt{R}$$

sont synonymes en LEAN : on n'est pas obligé de mettre les parenthèses précédentes.

```
theorem exemple_2_version_courte :
  ∀ x y : Int, (x = 3)  →  (y = 2 )  →  (x + y = 5) := by
{
  intro x y H1 H2
  rw [H1, H2]
  simp
}
```

Passer d'une écriture de la forme « $(P$ et $Q) \Rightarrow R$ » à une écriture de la forme « $P \Rightarrow (Q \Rightarrow R)$ » est une opération logique qui s'appelle la « curryfication » du nom du logicien et mathématicien américain Haskell Curry (1900-1982).

# Appliquer une hypothèse à des objets ou d'autres hypothèses

On a vu dans l'aide-mémoire que si on a dans le contexte LEAN :

- un énoncé `H` de la forme

$$\texttt{H : } \forall\ \texttt{z1 z2 ... zn : E, P(z1, ... )},$$

- des objets `w1`, `w2`, ... `wn` de type `E`,

alors on peut appliquer l'énoncé `H` aux objets `w1`, `w2`, ... `wn` pour obtenir une preuve de `P(w1, w2 ... wn)`, en tapant par exemple :

```
have H1 = H w1 w2 ... wn
```

De même, si on a dans le contexte :

- un énoncé `H` de la forme

$$\texttt{H : P} \to \texttt{Q},$$

- un énoncé `H1` affirmant que `P` est vraie :

$$\texttt{H1 : P}$$

alors on peut appliquer l'énoncé `H` à l'hypothèse `H1`, en tapant par exemple :

```
have H1 = H H1
```

```
theorem un_exemple :
  (∀ x y : Nat, x + y = 1 → (x = 0 ∨ y = 0 )) → ¬ (2 + 3 = 1) := by
{
    intro H1
    intro H_absurde
    have H2 := H1 2 3
    have H_deux_ou_trois_egal_zero := H2 H_absurde
    contradiction
}
```

En fait, on peut faire les deux opérations précédentes en une fois :

```
theorem un_exemple_v2 :
  (∀ x y : Nat, x + y = 1 → (x = 0 ∨ y = 0 )) → ¬ (2 + 3 = 1 ) := by
{
    intro H1
    intro H_absurde
    have H_deux_ou_trois_egal_zero := H1 2 3 H_absurde
    contradiction
}
```

Cela permet d'aller un peu plus vite : en une seule fois, on applique l'hypothèse H1 à 2, 3 et H_absurde.

# Comment fonctionne LEAN ?

Comme la plupart des assistants de preuve, LEAN repose sur l'utilisation de la **correspondance de Curry-Howard** entre preuves et programmes.

Il serait un peu hors de portée à notre niveau de faire une présentation rigoureuse de la façon dont cela est défini précisément, mais il peut néannmoins être utile de voir ce qui se passe dans la machine quand on utilise LEAN.

On va avoir besoin de faire quelques rappels avant cela.

# Rappels - Déclaration de variables

1. Pour déclarer une variable en LEAN, on utilise la syntaxe suivante :

   ```
   def nom_de_la_variable : type_variable := valeur_variable
   ```

   Par exemple :

```
def nombre_mystere : Nat := 5050.
```

2. Dans la mémoire, les variables sont stockées dans un tableau comme celui-ci :

| Nom de la variable | Type de la variable | Valeur de la variable |
|---|---|---|
| nombre_mystere | Nat | 5050 |
| ... | ... | ... |

3. On peut afficher le **type** de la variable avec `#check` et sa **valeur** avec `#print`.

| Commandes | Messages de la console |
|---|---|
| `#check nombre_mystere` | `nombre_mystere : Nat` |
| `#print nombre_mystere` | `5050` |

# Rappels - Déclaration de fonctions

1. Pour déclarer une fonction en LEAN, on peut utiliser la syntaxe suivante :

```
def nom_de_la_fonction : type_entree → type_sortie :=
                fun nom_entree => expression_sortie
```

   Par exemple :

```
def fonction_carre : Float→Float :=
                fun x => x^2
```

2. Dans la mémoire, les fonctions sont stockées dans le même tableau que les autres variables, mais leur type indique qu'il s'agit de fonctions. La valeur a aussi une écriture particulière, utilisant le mot-clé `fun` :

| Nom de la variable | Type de la variable | Valeur de la variable |
|---|---|---|
| `fonction_carre` | `Float→Float` | `fun x => x^2` |
| ... | ... | ... |

3. On peut appliquer une fonction à un objet du bon type, en écrivant le nom de cet objet après le nom de la fonction. On peut afficher le résultat avec `#eval`.

| Commandes | Messages de la console |
|---|---|
| `#eval (fonction_carre 32.1)` | 1030.410000 |

# Quel est le type d'un théorème LEAN ?

Définissons un théorème en LEAN, par exemple celui-ci :

```
theorem theoreme_facile :
  ∀ x : Nat, x = 1 → x + 1 = 2
:= by
{
    intro x H
    rw [H]
}
```

Puisque « `theoreme_facile` » est un objet défini dans le langage, il doit avoir un type et une valeur ! Essayons de les afficher avec `#check` et `#print`.

```
theorem theoreme_facile :
  ∀ x : Nat, x = 1 → x + 1 = 2
:= by
{
    intro x H
    rw [H]
}
```

- **Affichage du type : commande `#check`**

| **Commandes** | **Messages de la console** |
|---|---|
| `#check (theoreme_facile)` | `theoreme_facile : ∀ (x : ℕ) , x = 1 → x + 1 = 2` |

LEAN nous indique que « `theoreme_facile` » a pour **type** l'expression suivante :

$$\forall\ (x : \mathbb{N})\ ,\ x = 1 \to x + 1 = 2$$

```
theorem theoreme_facile :
  ∀ x : Nat, x = 1 → x + 1 = 2
:= by
{
    intro x H
    rw [H]
}
```

- **Affichage de la valeur : commande `#print`**

| Commandes | Messages de la console |
|---|---|
| `#print theoreme_facile` | `theoreme_facile : ∀ (x : ℕ) , x = 1 → x + 1 = 2 :=`<br>`fun x H`<br>`=> Eq.mpr (id (congrArg (fun _a => _a + 1 = 2) H)) (Eq.refl (1 + 1))` |

LEAN nous indique que « `theoreme_facile` » a pour **valeur** l'expression suivante :

```
fun x H =>
  Eq.mpr (id (congrArg (fun _a => _a + 1 = 2) H)) (Eq.refl (1 + 1))
```

LEAN nous indique ainsi que pour lui, « `theoreme_facile` » est en fait une fonction !

Avant d'expliquer cela, remarquons que ce qui précède signifie que les théorèmes sont stockés dans la mémoire de l'ordinateur aux côtés des autres variables et fonctions.

On a encore le tableau suivant :

```
def  nombre_mystere  :  Nat  :=  5050

def  fonction_interessante : Nat → Int :=
    fun x → x^2

theorem theoreme_facile :
  ∀ x : Nat, x =  1  →  x  + 1 = 2 := by
{
    intro x H
    rw [H]
}
```

| Nom de la variable | Type de la variable | Valeur de la variable |
|---|---|---|
| nombre_mystere | Nat | 5050 |
| fonction interessante | Nat → Int | fun x → x^2 |
| theoreme_facile | ∀ (x : ℕ) , x = 1 → x + 1 = 2 | fun x H => Eq.mpr (id (congrArg (fun _a => _a + 1 = 2) H)) (Eq.refl (1 + 1)) |

Les trois syntaxes sont en fait essentiellement les mêmes !

```
def  nombre_mystere   :  Nat   :=  5050

def  fonction_interessante  : Nat → Int :=
    fun x → x^2

theorem  theoreme_facile :
  ∀ x : Nat, x =  1  →  x  + 1 = 2 := by
{
    intro x H
    rw [H]
}
```

| Nom de la variable | Type de la variable | Valeur de la variable |
|---|---|---|
| nombre_mystere | Nat | 5050 |
| fonction interessante | Nat → Int | fun x → x^2 |
| theoreme_facile | ∀ (x : ℕ) , x = 1 → x + 1 = 2 | fun x H => Eq.mpr (id (congrArg (fun _a => _a + 1 = 2) H)) (Eq.refl (1 + 1)) |

Pourquoi un théorème serait-il encodé comme une fonction dans l'ordinateur ?

Cette idée repose sur la **correspondance de Curry-Howard**, aussi appelée « correspondance preuves-programmes ».

L'idée fondamentale est que pour l'ordinateur, la **valeur** d'un théorème est un **programme** qui démontre son **énoncé**, qui est représenté par le **type du programme.**

| **Nom de la variable** | **Type de la variable** | **Valeur de la variable** |
|---|---|---|
| `theoreme_facile` | `∀ (x : ℕ) , x = 1 → x + 1 = 2` | `fun x H =>`<br>`Eq.mpr (id (congrArg (fun _a => _a + 1 = 2) H))`<br>`(Eq.refl (1 + 1))` |
| Nom du théorème | Enoncé du théorème | Programme prouvant l'énoncé |

En fait, on peut utiliser les expressions « Preuve de l'énoncé » et « Programme prouvant l'énoncé » de façon un peu interchangeable.

De quels types de programme peut-il s'agir ? Pour les propositions du type suivant, c'est assez simple à voir :

| | **Enoncé** | **Programme correspondant** |
|---|---|---|
| **Implication** | $P \Rightarrow Q$ | • *Fonction prenant en entrée une preuve de* `P`*, et renvoyant une preuve de* `Q`*.* |
| **Quantificateur** $\forall$ | $\forall x \in E, P(x)$ | • *Fonction prenant en entrée un objet* `x` *de type* `E`*, et renvoyant en sortie une preuve de* `P x` |
| **Quantificateur** $\exists$ | $\exists x \in E, P(x)$ | • *Programme sur deux « cases mémoires » :*<br>1. *une « case mémoire » contenant un objet* `x` *de type* `E` *;*<br>2. *une « case mémoire » contenant une preuve de* `P x` |

Pour les propositions $P$ et $Q$, $P$ ou $Q$, $\neg P$, il y a aussi une interprétation en termes de programmes, mais un peu plus élaborées.

# Ecrire une preuve comme si on écrivait une fonction : un exemple

Démontrons de deux manières différentes que pour toutes propositions $P, Q, R$, on a la propriété

$$P \Longrightarrow \big((P \Longrightarrow Q) \Longrightarrow \big((Q \Longrightarrow R) \Longrightarrow R\big)\big)$$

*« Pour toutes propositions $P, Q, R$,* **si** *$P$ est vraie,* **alors si** *$P$ implique $Q$,* **alors si** *$Q$ implique $R$,* **alors** *$R$ est vraie. »*

**Preuve en LEAN en suivant l'aide-mémoire :**

```
theorem  enonce_preuve_1 :
∀ P Q R :  Prop, P → (P → Q) → (Q → R) → R := by
{
  intro P Q R
  intro H1
  intro H2
  intro H3
  have H4 := H2 H1
  have H5 := H3 H4
  assumption
}
```

**Preuve en LEAN comme si on écrivait une fonction :**

```
theorem  enonce_preuve_2 :
∀ P Q R :  Prop, P → (P → Q) → (Q → R) → R :=
    fun P Q R H1 H2 H3 => H3 (H2 H1)
```

La fonction prend en entrée :

1. trois propositions, notées `P`, `Q`, `R` ci-dessus ;
2. une preuve de `P`, notée `H1` ;
3. une preuve de `P` → `Q`, notée `H2` ;
   *(c'est littéralement une* fonction *qui prend une preuve de* `P` *et renvoie une preuve de* `Q`*)*
4. une preuve de `Q` → `R`, notée `H3` ;
   *(c'est littéralement une* fonction *qui prend une preuve de* `Q` *et renvoie une preuve de* `R`*)*

La fonction que nous avons écrite renvoie en sortie une preuve de `R`, en appliquant `H2` à `H1`, puis en donnant le résultat à `H3`.

La première version `enonce_preuve_1` a été écrite avec le mot-clé `by`, pour indiquer à LEAN que l'on allait construire une preuve en utilisant l'approche par *tactiques*. L'ordinateur convertit ensuite les arguments de la preuve en une fonction.

On peut afficher les deux fonctions obtenues avec la commande `#print` :

| **Commandes** | **Messages de la console** |
|---|---|
| `#print enonce_preuve_1` | `enonce_preuve_1 : ∀ P Q R : Prop , P → (P → Q) → (Q → R) → R :=`<br>`  fun P Q R H1 H2 H3 =>`<br>`  let_fun H4 := H2 H1;`<br>`  let_fun H5 := H3 H4;`<br>`  H5` |
| `#print enonce_preuve_2` | `enonce_preuve_2 : ∀ P Q R : Prop , P → (P → Q) → (Q → R) → R :=`<br>`  fun P Q R H1 H2 H3 => H3 (H2 H1)` |

Dans le deuxième cas, c'est littéralement ce qu'on avait tapé !

Dans le premier cas, la preuve a été convertie en une fonction un peu plus compliquée.

Les preuves écrites avec l'approche par tactique (mot-clé `by`) sont donc une version plus facilement compréhensible par un être humain, que l'ordinateur convertit en un programme qu'il peut comprendre.

Parfois, le programme peut-être extrêmement compliqué, alors que la preuve est très courte !

C'est typiquement ce qui se produit quand on utilise des tactiques de simplication, comme `simp` ou `linarith`.

Par exemple :

```
theorem  addition_commutative_1 :
  ∀ x y  : Int, x + y = y + x := by
{
    intro  x y
    linarith
}
```

Le résultat affiché avec `#print` est très compliqué !

```
theorem addition_commutative_1 : ∀ (x y : ℤ), x + y = y + x :=
fun x y ⇒
  Linarith.eq_of_not_lt_of_not_gt (x + y) (y + x)
    (Not.intro fun a ⇒
      Linarith.lt_irrefl
        (Eq.mp
          (congrArg (fun _a ⇒ _a < 0)
            (Mathlib.Tactic.Ring.of_eq
              (Mathlib.Tactic.Ring.add_congr
                (Mathlib.Tactic.Ring.neg_congr
                  (Mathlib.Tactic.Ring.cast_pos (Mathlib.Meta.NormNum.isNat_ofNat ℤ (Eq.refl 1)))
                  (Mathlib.Tactic.Ring.neg_add
                    (Mathlib.Tactic.Ring.neg_one_mul
                      (Mathlib.Meta.NormNum.IsInt.to_raw_eq
                        (Mathlib.Meta.NormNum.isInt_mul (Eq.refl HMul.hMul)
                          (Mathlib.Meta.NormNum.IsInt.of_raw ℤ (Int.negOfNat 1))
                          (Mathlib.Meta.NormNum.IsNat.to_isInt (Mathlib.Meta.NormNum.IsNat.of_raw ℤ 1))
                          (Eq.refl (Int.negOfNat 1)))))
                    Mathlib.Tactic.Ring.neg_zero))
                (Mathlib.Tactic.Ring.sub_congr
                  (Mathlib.Tactic.Ring.add_congr
                    (Mathlib.Tactic.Ring.add_congr (Mathlib.Tactic.Ring.atom_pf x) (Mathlib.Tactic.Ring.atom_pf y)
                      (Mathlib.Tactic.Ring.add_pf_add_lt (x ^ Nat.rawCast 1 * Nat.rawCast 1)
                        (Mathlib.Tactic.Ring.add_pf_zero_add (y ^ Nat.rawCast 1 * Nat.rawCast 1 + 0))))
                    (Mathlib.Tactic.Ring.cast_pos (Mathlib.Meta.NormNum.isNat_ofNat ℤ (Eq.refl 1)))
                    (Mathlib.Tactic.Ring.add_pf_add_gt (Nat.rawCast 1)
                      (Mathlib.Tactic.Ring.add_pf_add_zero
                        (x ^ Nat.rawCast 1 * Nat.rawCast 1 + (y ^ Nat.rawCast 1 * Nat.rawCast 1 + 0)))))
                  (Mathlib.Tactic.Ring.add_congr (Mathlib.Tactic.Ring.atom_pf y) (Mathlib.Tactic.Ring.atom_pf x)
                    (Mathlib.Tactic.Ring.add_pf_add_gt (x ^ Nat.rawCast 1 * Nat.rawCast 1)
                      (Mathlib.Tactic.Ring.add_pf_add_zero (y ^ Nat.rawCast 1 * Nat.rawCast 1 + 0))))
                  (Mathlib.Tactic.Ring.sub_pf
                    (Mathlib.Tactic.Ring.neg_add
                      (Mathlib.Tactic.Ring.neg_mul x (Nat.rawCast 1)
                        (Mathlib.Tactic.Ring.neg_one_mul
                          (Mathlib.Meta.NormNum.IsInt.to_raw_eq
                            (Mathlib.Meta.NormNum.isInt_mul (Eq.refl HMul.hMul)
                              (Mathlib.Meta.NormNum.IsInt.of_raw ℤ (Int.negOfNat 1))
                              (Mathlib.Meta.NormNum.IsNat.to_isInt (Mathlib.Meta.NormNum.IsNat.of_raw ℤ 1))
                              (Eq.refl (Int.negOfNat 1))))))
                      (Mathlib.Tactic.Ring.neg_add
                        (Mathlib.Tactic.Ring.neg_mul y (Nat.rawCast 1)
                          (Mathlib.Tactic.Ring.neg_one_mul
                            (Mathlib.Meta.NormNum.IsInt.to_raw_eq
                              (Mathlib.Meta.NormNum.isInt_mul (Eq.refl HMul.hMul)
```

En fait, cela vient de ce que la commande `#linarith` essaie d'appliquer un très grand nombre de théorèmes de la bibliothèque pour obtenir le résultat, ce qui n'est pas très optimal !

Heureusement, nous n'aurons pas trop à nous préoccuper de cela : pour nous, ces choses se produisent de manière cachée dans l'ordinateur.