Initiation à la preuve avec L∃∀N

Esisar 1A - P2027

F. Tran-Minh

12 décembre 2022

1 INTRODUCTION

Au cours de leur histoire, les mathématiques ont consolidé leur cohérence interne en proposant des théories axiomatiques : un petit nombre d'énoncés est accepté, sans preuve : ce sont les axiomes ; et les autres résultats ne sont acceptés commes valides, et dès lors nommés théorèmes, que s'ils se déduisent des axiomes en utilisant un nombre limité de règles de logique, ou de réécriture. Le texte déductif y conduisant en constitue la preuve.

Historiquement, les théories axiomatiques ont pu concerner un domaine particulier des mathématiques. La plus connue, la géométrie Euclidienne (300 ans av JC) déduit 13 volumes de théorèmes de seulement 5 axiomes. Ces 5 axiomes sont acceptés en tant que tels parce qu'ils décrivent une vérité dans un modèle réel, physique. Exemple : "un segment de droite peut être tracé en joignant deux points quelconques distincts." Le petit nombre de ces axiomes rassure sur le risque encouru en les admettant.

Plus récemment, des théories permettant de déduire et d'unifier la grande majorité des mathématiques usuelles ont vu le jour ; elles énoncent des axiomes sur des objets plus abstraits. Une des plus usitées est la théorie ZFC (Zermelo-Fraenkel avec axiome du choix, à partir de 1908), dont les objets de base sont les ensembles. [1] Seulement 10 axiomes (dont deux schémas d'axiomes) permettent de déduire l'ensemble des mathématiques usuelles. Par exemple, l'un de ces axiomes (l'axiome de la paire) énonce que si a et b sont deux ensembles alors il existe un ensemble c contenant exactement a et b. Le nombre réduit d'axiomes réduit le risque d'inconsistance de la théorie (l'existence de contradiction).

Entrer dans l'activité mathématique, c'est adhérer à ce paradigme : chaque nouvelle affirmation (quel qu'ait été le processus de sa découverte) devra, pour être validée dans la rationnalité mathématique, et acquérir le statut de théorème, se déduire d'axiomes ou de théorèmes (ayant été établis précédemment de même), selon quelques règles précises de logique et de réécriture. Par exemple "pour prouver l'affirmation « A ET B » il suffira, dans l'ordre que vous voulez, de présenter une preuve de A et une preuve de B". La preuve a de fait une fonction sociale - elle permet de convaincre, et de se convaincre, qu'un résultat est correct - et une fonction pédagogique - elle permet de mieux comprendre le "fonctionnement" d'un théorème, ce qu'il dit exactement, les raisons pour lesquelles telle hypothèse est utile ou pourquoi la conclusion est formulée d'une certaine façon.

Dans ce module, on propose d'utiliser des assistants de preuve ou des logiciels qui en sont dérivés, pour s'approprier tant l'enjeu que les exigences et les techniques de la preuve en mathématiques. Les étudiant-e-s interfereront avec ces logiciels soit par le moyen d'une interface graphique (logiciel Edukera), soit par l'apprentissage d'un langage informatique spécifique (logiciel L∃∀N). La possibilité d'une rétroaction instantanée de la machine, couplée à celle du travail en autonomie permettant à chacun-e d'avancer à son rythme permettra, on l'espère, de stimuler l'apprentissage.

Les contraintes de l'interface d'Edukera et la syntaxe de L∃∀N miment les exigences syntaxiques du formalisme mathématique. Néanmoins, une démonstration mathématique - papier - n'est ni une série de clics ni un texte en langage informatique. Par contre l'un (papier) et l'autre (numérique) partageront la même structure, mais pas forcément le même niveau de détails, ni la même hierarchie des idées à mettre en valeur : alors que deux arguments pourraient être homogènes pour le logiciel, le professeur lisant une copie papier trouverait que l'un ne vaut pas la peine d'être souligné alors que le second est crucial au vu du cours qu'il a donné la veille... Ce passage entre la preuve informatique et la preuve papier est donc un exercice à part entière qu'on veillera à travailler spécifiquement et systématiquement.

1.1 *Objectifs*

- Consolider sa perception de la nature des objets
- Consolider sa maitrise de la syntaxe du langage formalisé mathématique
- Consolider sa perception du rôle de la preuve
- Construire des automatismes pour élaborer/ construire / rédiger des preuves correctes
- Espace plus informel pour échanger autour de la notion de preuve

1. Cependant les assistants de preuve utilisés dans ce module s'appuient sur une autre théorie axiomatique puissante appelée théorie des Types.

2 ENVIRONNEMENT DE TRAVAIL

2.1 *Edukera*

Le système s'utilise dans le navigateur :

— Accédez à : `https://app.edukera.com/`

— Créez un compte avec votre adresse Esisar

2.2 *L∃∀N*

Attention nous travaillerons avec L∃∀N 3 qui n'est pas compatible avec L∃∀N 4...
Le système peut être utilisé dans le navigateur :

- Accédez à `https://leanprover-community.github.io/lean-web-editor`

Le système peut être installé et utilisé localement [il est déjà installé sur les machines en B141] :

- Système recommandé : Linux (fonctionne aussi sur d'autres systèmes)
- L'interface de développement suggérée est Visual Studio Code [`https://code.visualstudio.com`]
- On peut aussi utiliser emacs
- L'installation de L∃∀N 3 est décrite ici : `https://leanprover-community.github.io/install/linux.html`

2.3 *Références internet*

— `https://leanprover-community.github.io/mathlib_docs/`

— `https://leanprover.github.io/reference/`

3.1 *Pas à pas : un premier théorème*

Dans Lean Web Editor ou dans Visual Studio Code, entrons le premier théorème ci-dessous :

```
-- Ceci est un commentaire: ce texte est ignoré par le compilateur

theorem mon_premier_theoreme : ∀ P:Prop, P → P  :=          -- nom du théorème : énoncé du théorème := preuve du théorème
  assume (P:Prop),                                          -- Le symbole ∀ s'obtient en tapant \all puis espace
    assume(h1 : P),                                         -- Le symbole → s'obtient en tapant \r puis espace
      show P, from h1                                       -- Voir l'annexe pour tous les raccourcis
```

Remarquez que l'implication, qui se note avec une double flèche dans un texte mathématique, est ici représentée par une simple flèche. Par ailleurs, il est très fortement recommandé de s'imposer une indentation (décalage vers la droite du texte avec des espaces) qui traduit la structure de la preuve, bien que L∃∀N ne sera pas sensible aux espacements et aux retours à la ligne, voire même des parenthèses supplémentaires pour clarifier la structure :

En L∃∀N :

```
theorem mon_premier_theoreme : ∀ P:Prop, P → P  :=
  assume (P:Prop),
  (
    assume(h1 : P),
    (
      show P, from h1
    )
  )
```

En Maths :

Théorème (Mon Premier Théorème)

Pour toute assertion P, on a $P \Longrightarrow P$.

Démonstration. Soit P une assertion.
Supposons P.
Alors d'après la ligne précédente, P est vraie.
□

Sauvegardez ce fichier sous le nom `votrelogin_000.lean`. (Dans Lean Web Editor, il sauvegarde par défaut dans `_test.lean` il est donc recommandé de choisir "ouvrir le fichier" avec un éditeur (au lieu d'enregistrer) puis de sauvegarder sous le nom voulu.)
En promenant le curseur dans la preuve, on peut observer l'évolution

— Du contexte : les variables et hypothèses dont on dispose en un point donné de la preuve ;

— Du but (goal) local : ce qu'il s'agit de prouver en un point donné de la preuve.

Si on veut seulement énoncer le théorème, et qu'on reporte la preuve à plus tard, on peut remplacer la preuve par `sorry`, ce qui génèrera un "Warning" :

```
theorem mon_premier_theoreme : ∀ P:Prop, P → P  :=  sorry
```

Remarque : dans cet exemple et les quelques exemples suivants, on utilise le connecteur ∀ pour quantifier des objets de type "assertion". Cette pratique n'est pas usitée ni en général acceptée dans un cours standard de mathématiques de 1er cycle, parce que les assertions ne sont pas des objets mathématiques à proprement parler, mais "méta mathématiques" (la théorie qui parle des maths). En maths, on quantifie des objets mathématiques : nombres, vecteurs, ensembles... mais pas des assertions. Dans L∃∀N , cette généralisation/extension élégante et puissante va en outre nous faciliter l'apprentissage.

3.2 *Notion d'introduction / élimination*

On distingue trois façons d'interagir avec une assertion mathématique : on peut vouloir :

— ... l'énoncer

— ... l'utiliser, en supposant qu'elle a déjà été prouvée avant [on parle d'élimination du connecteur principal]

— ... la prouver [on parle d'introduction du connecteur principal]

Dans le théorème ci-dessus nous avons deux connecteurs : ∀ et →.
Voici les règles d'introduction et d'élimination de ces connecteurs.

3.3 *Introduction / élimination du quantificateur universel*

Connecteur ∀ : **Règles** d'introduction/élimination		
	L∃∀N	Maths
Enoncé	`Assertion1: (∀x:E,P x)`	(1) $\forall x \in E, P(x)$
Introduction (pour prouver...) Le mot clé "Soit" des maths se traduit par `assume` en L∃∀N .	`assume(x:E),` `  show (P x), from sorry` ou : `assume(x:E), (sorry:(P x))` (`sorry` est remplacé par une preuve adéquate)	Soit $x \in E$. Prouvons $P(x)$: ... ... donc $P(x)$.
Elimination (pour utiliser...)	`example (y:E) : P y := Assertion1 y`	Etant donné $y \in E$, on a $P(y)$. Pour le prouver, il suffit d'appliquer l'assertion (1) à y qui est bien un élément de E.

Connecteur ∀ : **Exemple** d'introduction/élimination		
	L∃∀N	Maths
Enoncé	`Assertion1: (∀x:ℝ,x^2 ⩾0)`	(1) $\forall x \in \mathbb{R}, x^2 \geqslant 0$
Introduction (pour prouver...)	`assume(x:ℝ),` `  show (x^2 ⩾ 0), from sq_nonneg x`	Soit $x \in \mathbb{R}$. si $x \geqslant 0$ alors $x^2 = x \cdot x \geqslant 0$; si $x < 0$, ... donc $x^2 \geqslant 0$.
Elimination (pour utiliser...)	`example (y:ℝ) : y^2-2*y+1 ⩾ 0 :=` `  calc` `  y^2-2*y+1 = (y-1)^2 : by ring` `  ...  ⩾ 0  : Assertion1 (y-1)` `  -- élimination: on applique` `  Assertion1 au réel (y-1)`	Etant donné $y \in \mathbb{R}$, en appliquant l'assertion (1) à $y-1$, qui est bien un nombre réel, on obtient $(y-1)^2 \geqslant 0$. Or $y^2 - 2y + 1 = (y-1)^2$. Donc $y^2 - 2y + 1 \geqslant 0$.

3.4 *Introduction / élimination de l'implication*

Connecteur → : **Règles** d'introduction/élimination		
	L∃∀N	Maths
Enoncé	`Assertion2: ( P → Q)`	(2) $P \Longrightarrow Q$
Introduction (pour prouver...) Le mot clé "Supposons" des maths se traduit par `assume` en L∃∀N .	`assume (h1: P),` `  show Q, from sorry` ou bien : `assume(h1: P), (sorry:Q)` (`sorry` est remplacé par une preuve adéquate)	Supposons P. ... ... donc Q.
Elimination (pour utiliser...)	`example (h1:P) : Q := Assertion1 h1 -- si on` `  dispose d'une preuve h1 de P il suffit` `  de lui appliquer Assertion1`	On considère ici qu'on a déjà établi que l'assertion P est vraie. En appliquant (2), on obtient donc Q.

Exercice 3.4.1 (`votrelogin_341.lean`)

1. Dans Lean, énoncez puis démontrez le théorème suivant "Pour toutes propositions P et Q, $P \Longrightarrow (Q \Longrightarrow P)$".
2. Ecrivez une preuve papier de ce théorème.

4.1 *La conjonction ($\wedge$, ET)*

Le ET est noté $\wedge$.
Voici ses règles d'introduction/élimination :

Connecteur $\wedge$: **Règles** d'introduction/élimination		
	L∃∀N	Maths
Enoncé	`Assertion1: P ∧ Q`	(1) P ET Q
Introduction (pour prouver...) Pour prouver P $\wedge$ Q, il suffit de donner une preuve de P ...et... une preuve de Q ! On assemble les deux à l'aide de `and.intro`	Sous les hypothèses `h1: P` et `h2: Q`, `and.intro h1 h2` est une preuve de `P ∧ Q` .	... blabla ... donc on obtient P. ... blabla ... d'où on a aussi Q. Ainsi, P ET Q est vraie.
Elimination (pour utiliser...) Il y a deux façons d'utiliser P $\wedge$ Q. On peut décider d'exploiter P (élimination gauche) ou bien d'exploiter Q (élimination droite).	Sous l'hypothèse `Assertion1: P ∧ Q` : `Assertion1.left` est une preuve de `P` `Assertion1.right` est une preuve de `Q`	Comme P ET Q, on a P. Comme P ET Q, on a Q.

Exemple 1

```
theorem T2 : ∀ P Q:Prop, P → (Q → P ∧ Q) :=
  assume (P Q:Prop),
  (
    assume(h1 : P),
    (
      assume(h2 : Q),
      (
         (and.intro h1 h2):  P ∧ Q
      )
    )
  )
```

Remarque : dans L∃∀N , l'opérateur d'implication $\rightarrow$ a la propriété particulière de s'associer à droite, c'est-à-dire que le théorème précédent aurait pu s'énoncer `∀P Q:Prop, P → Q → P ∧ Q`. Par ailleurs une assertion de la forme `P → Q → R` signifie : "Si P est vraie alors (Si Q est vraie alors R est vraie)" ce qui est équivalent à " Si P et Q sont vraies alors R est vraie". La première écriture est moins naturelle mais, on le verra, elle est plus pratique à manier dans L∃∀N .

Exemple 2

```
theorem T3 : ∀ P Q : Prop, P ∧ Q → P :=
   assume (P Q :Prop) ,
   (
     assume (h: P ∧ Q),
     (
       h.left : P
     )
   )
```

Exercice 4.1.2 (`votrelogin_412.lean`)

1. Dans Lean, énoncez puis démontrez le théorème suivant :
   "Pour toutes propositions P et Q, si P $\wedge$ Q alors Q".
2. Ecrivez une preuve papier de ce théorème.
3. Dans Lean, énoncez puis démontrez le théorème suivant :
   "Pour toutes propositions P et Q, si P $\wedge$ Q alors Q $\wedge$ P".
4. Ecrivez une preuve papier de ce théorème.

4.2 *La disjonction ($\vee$, OU)*

Le OU est noté $\vee$.
Voici ses règles d'introduction/élimination :

Connecteur $\vee$: **Règles** d'introduction/élimination		
	L∃∀N	Maths
Enoncé	`Assertion1: P ∨ Q`	(1) P OU Q
Introduction (pour prouver...) Il y a deux introductions possibles pour $\vee$: soit à gauche (avec P) : on utilise `or.inl`, soit à droite (avec Q) : on utilise `or.inr`,	[A GAUCHE] Sous l'hypothèse `h1: P`, `or.inl h1` est une preuve de `P ∨ Q` . [A DROITE] Sous l'hypothèse `h2: P`, `or.inr h2` est une preuve de `P ∨ Q` .	[A GAUCHE] . . . Comme P est vraie, on a P OU Q. [A DROITE] . . . Comme Q est vraie, on a P OU Q.
Elimination (pour utiliser...) Pour utiliser P OU Q, on réalise une disjonction de cas : on traite le cas où P serait vraie, et le cas où ce serait Q qui est vraie. On doit aboutir à une conclusion commune. Les deux cas sont introduits par `or.elim`.	Voici une preuve d'une assertion `R` sous l'hypothèse `Assertion1: P ∨ Q` : `or.elim Assertion1` `( assume(h1:P), (sorry:R) )` `( assume(h2:P), (sorry:R) )` (les `sorry` sont remplacés par des preuves adéquates)	(...) Ainsi P $\vee$ Q est donc vraie. Procédons par disjonction de cas : — Si P est vraie, alors .... donc R ; — Si Q est vraie, alors .... donc R ; Dans tous les cas, on a R.

Exemple

```
theorem T4 : ∀ P Q : Prop, P → P ∨ Q :=
  assume (P Q :Prop) ,
  (
    assume (h: P),
    (
      (or.inl h ): P ∨ Q
    )
  )
```

Exercice 4.2.3 (`votrelogin_423.lean`)

1. Dans Lean, énoncez puis démontrez le théorème suivant :
   "Pour toutes propositions P et Q, si P $\wedge$ Q alors P $\vee$ Q".
2. Ecrivez une preuve papier de ce théorème.

4.3 *La négation*

La négation $\neg$P d'une assertion P est définie comme l'assertion `P → false` c'est-à-dire qu'une assertion est fausse si et seulement si on peut en déduire une contradiction.
Voici ses règles d'introduction/élimination :

Connecteur $\vee$: **Règles** d'introduction/élimination		
	L∃∀N	Maths
Enoncé	`Assertion1: ¬P  --tapez \not`	(1) NON P
Introduction (pour prouver...)	`assume (h1:P),` `show false, from sorry` (`sorry` est remplacé par une preuve adéquate)	Supposons P vraie. Alors · · · . · · · · · · Absurde. donc P est fausse.
Elimination (pour utiliser...)	Sous l'hypothèse `Assertion2: P` , l'élimination : `absurd Assertion2 Assertion1` est une preuve de ce qu'on veut.	P étant fausse, telle hypothèse qui conduit à P ne peut pas être valide.

Exercice 4.3.4 (votrelogin_434.lean)

1. On considère le texte de preuve Lean ci - dessous :

```
constants IlFaitBeau IlPleut JeprendsMonParapluie : Prop

axiom a1: IlFaitBeau ∨ IlPleut
axiom a2: IlPleut → JeprendsMonParapluie

theorem T5 : ¬ IlFaitBeau → JeprendsMonParapluie := sorry
```

   On observe qu'il définit trois assertions : IlFaitBeau, IlPleut, JeprendsMonParapluie qui sont arbitrairement liées par les axiomes suivants :

   (a1) En tous temps, soit il fait beau, soit il pleut

   (a2) Si il pleut, je prends mon parapluie.

   Remplacez le `sorry` du théorème T5 par une preuve adéquate, justifiant que si il ne fait pas beau, je prends mon parapluie.

2. Ecrivez une preuve papier de ce théorème.

4.4 *L'équivalence*

Vous l'avez vu en cours de maths, prouver que P et Q sont équivalentes, c'est prouver que P implique Q et que Q implique P. On retrouve cette introduction dans L∃∀N . A noter que l'équivalence, notée $\iff$ en maths, est notée $\leftrightarrow$ dans L∃∀N .

Voici ses règles d'introduction/élimination :

<table>
<tr><th colspan="3">Connecteur ↔ : Règles d'introduction/élimination</th></tr>
<tr><th></th><th>L∃∀N</th><th>Maths</th></tr>
<tr><td>Enoncé</td><td><code>Assertion1: P ↔Q</code></td><td>(1) P $\iff$ Q</td></tr>
<tr><td>Introduction
(pour prouver...)</td><td>Sous les hypothèses <code>h1: P →Q</code> et <code>h2: Q →P</code>,
<code>iff.intro h1 h2</code>
est une preuve de <code>P ↔Q</code> . On peut remplacer <code>h1</code> et <code>h2</code> en important directement l'introduction de → :
<pre>iff.intro
  (
    assume (hP:P),
    sorry: Q
)   -- : P→Q
  (
    assume (hQ:Q),
    sorry: P
)   -- : Q→P</pre></td><td>(a) Prouvons P $\implies$ Q.
Supposons P.
. . .
.. donc Q.
(b) Prouvons Q $\implies$ P.
Supposons Q.
. . .
.. donc P.</td></tr>
<tr><td>Elimination
(pour utiliser...)</td><td>Sous l'hypothèse <code>Assertion1: P ↔Q</code> ,
<code>Assertion1.mp</code> est une preuve de <code>P →Q</code> .
<code>Assertion1.mpr</code> est une preuve de <code>Q →P</code> .
On peut ensuite appliquer si besoin l'élimination de → vue précédemment. Par exemple, sous l'hypothèse supplémentaire <code>hQ: Q</code>, le terme <code>Assertion1.mpr hQ</code> est une preuve de <code>P</code> .</td><td>• Comme P $\iff$ Q, on a donc P $\implies$ Q.
• Comme P $\iff$ Q, on a donc Q $\implies$ P.</td></tr>
</table>

Remarque : "mp" signifie "modus ponens".

Exercice 4.4.5 (votrelogin_445.lean)

1. Enoncez, puis démontrez dans Lean que pour toutes assertions P et Q, on a P ET Q $\iff$ Q ET P.
2. Ecrivez la preuve papier.
3. Enoncez, puis démontrez dans Lean que pour toutes assertions P et Q, on a P OU Q $\iff$ Q OU P.
4. Ecrivez la preuve papier.

Exercice 4.4.6 (votrelogin_446.lean)

Donnez une preuve L∃∀N du théorème ci-dessous :

```
theorem T  : ∀ (A B:Prop), (A ∨ B) ↔ ((A →B)→B) := sorry   -- vous pouvez utiliser (classical.em A) qui est une preuve de A∨¬A
```

Exercice 4.4.7 (votrelogin_447.lean)

On veut prouver :

$$\forall a \in \mathbb{R}, \forall b \in \mathbb{R}, ((\forall x \in \mathbb{R}, ax + b = 0) \iff (a = 0 \text{ ET } b = 0))$$

1. Ecrivez une preuve papier.
2. Ecrivez le squelette de la preuve dans L∃∀N (complétez par `sorry` pour les parties que vous ne savez pas faire).

(Voir l'exercice 5.1.9 pour compléter cette preuve)

Exercice 4.4.8 (votrelogin_448.lean)

Donnez une preuve L∃∀N du théorème ci-dessous :

```
theorem iff_iff_mp_mtmpr : ∀ P Q : Prop,  (P ↔ Q) ↔ ((P → Q) ∧ ( ¬ P →¬ Q)) :=  sorry
```

Vous pouvez utiliser les lemmes ci-dessous :

```
#check mt           -- permet d'obtenir la contraposée d'une implication
#check not_not      -- non non P équivaut à P
```

4.5 *Quantificateur existentiel*

∃ : **Règles** d'introduction/élimination		
	L∃∀N	Maths
Enoncé	`Assertion1: ∃x:E, P x`	(1)∃x ∈ E, P(x)
Introduction (pour prouver...)	Pour prouver `∃x:E, P x` , il faut présenter un élément `a` de type `E` et une preuve que `P a` est vraie : Etant donné `a:E`, et sous l'hypothèse `ha: P a` : `exists.intro a ha` est une preuve de `∃x:E, P x`.	(a) Prouvons ∃x ∈ E, P(x). Soit a := ···. (...) donc a est bien élément de E. De plus, ... donc P(a).
Elimination (pour utiliser...)	Voici une preuve d'une assertion `R` sous l'hypothèse `Assertion1: ∃x:E, P x` : `exists.elim Assertion1` `(` `  assume(x:E) (h: P x),` `  (sorry: R)` `)`	A ce stade on sait que ∃x ∈ E, P(x). Choisissons un tel x ∈ E vérifiant P(x). ... donc R.

Exemple 1

```
import data.nat.basic

#check mul_assoc

definition entier_pair (n:ℕ) : Prop := ∃ (k:ℕ), n = 2*k

theorem multiple_de_4_pair : ∀ k:ℕ, entier_pair (4*k) :=
  assume (k:ℕ ),
  (
    exists.intro
        (2*k)
        (calc
            4*k = (2*2)*k : eq.refl _
            ... = 2*(2*k) : mul_assoc 2 2 k
        )
  )
```

Pour comprendre les détails de la preuve ci-dessus, lire le passage sur l'environnement `calc` au début de la section suivante.

5.1 *Faire des calculs avec L∃∀N : **have**, **calc**, congr_arg*

***have** permet d'introduire un énoncé intermédiaire*

```
have nom_enonce : enonce, from preuve,
```

***calc** permet d'enchainer des calculs par transitivité*

```
calc
   formule1 = formule2   : preuve_que_formule1_egale_formule2
     ...    = formule3   : preuve_que_formule2_egale_formule3
     ...    = formule4   : preuve_que_formule3_egale_formule4
```

Le terme ci-dessus est une preuve que `formule1=formule4`. Cet environnement fonctionne aussi avec d'autres relations transitives, comme $\leqslant$.

congr_arg permet de remplacer une expression par une autre expression égale dans une troisième expression

Tout d'abord, une notation fondamentale en L∃∀N et dans tout langage fonctionne est λ. (Cette notation donne d'ailleurs son nom au λ-calcul). Il permet de signifier "la fonction qui à z associe... ". Il correspond à la flèche à talon $\mapsto$ utilisée en maths.
Exemple avec λ :

```
import data.real.basic
-- la ligne ci-dessous définit la fonction f qui à un réel x associe x+1
def f := λ x:ℝ, x+1

-- de façon équivalente, voici la fonction g   (égale  à f )
def g : ℝ→ℝ := λ x, x+1
```

Venons-en à `congr_arg`. Supposons que `h_uv` soit une preuve de `u=v` et que `f` soit une fonction.
Alors le terme :

```
congr_arg f  h_uv
```

est une preuve de `f u= f v`.

A propos des égalités : symm , trans et refl

Supposons que `h_uv` soit une preuve de `u=v` et que `h_vw` soit une preuve de `v=w`.
Alors `h_uv.symm` (qui peut aussi s'écrire `eq.symm h_uv`) est une preuve de `v=u`.
Par ailleurs, `eq.trans h_uv h_vw` est une preuve de `u=w`.
Notez enfin que le terme `eq.refl u` est une preuve de `u=u` mais qu'en pratique cela peut servir :

- à réduire des calculs numériques élémentaires sur les entiers ou les rationnels, par ex `eq.refl 4` est une preuve de `2*2=4`
- à prouver des égalités vraies par définition

Exemple

```
import data.real.basic

#check sub_self   -- sub_self : ∀ (a : ?M_1), a - a = 0
#check @sub_self  -- sub_self : ∀ {G : Type u_1} [_inst_1 : add_group G] (a : G), a - a = 0
#check add_sub    -- add_sub  : ∀ (a b c : ?M_1), a + (b - c) = a + b - c
#check @add_sub   -- add_sub  : ∀ {G : Type u_1} [_inst_1 : sub_neg_monoid G] (a b c : G), a + (b - c) = a + b - c

theorem T : ∀ a:ℝ,   a+1=1 →∀ x:ℝ, a*x = 0 :=
  assume a:ℝ,
    assume hyp1 : a+1=1,
      assume x:ℝ,

        have hyp2 : (1:ℝ) -1 = 0, from sub_self 1,
        have hyp3 : a=0, from
          calc
            a   = a+ 0        : (add_zero a).symm
            ... = a+(1-1)     : congr_arg (λ z:ℝ, a+z) hyp2.symm
            ... = a+1 - 1     : add_sub a 1 1
            ... = 1 -1        : congr_arg (λ z:ℝ , z-1) hyp1
            ... = 0           : hyp2
          ,
        calc
          a*x = 0*x : congr_arg (λ z:ℝ , z*x) hyp3
          ... = 0   : zero_mul x
```

Exercice 5.1.9 (votrelogin_5109.lean)

1. Donnez une preuve en L∃∀N du théorème suivant :

```
import data.real.basic
theorem T_cs : ∀ a:ℝ, ∀ b:ℝ, (  (a=0 ∧ b=0) → (∀ x:ℝ, a*x + b =0   )) := sorry
```

2. Complétez les parties **sorry** de votre réponse à l'exercice 4.4.7.

Vous pouvez utiliser librement les lemmes ci-dessous :

```
#check zero_add
#check add_zero

#check zero_mul
#check mul_zero
#check mul_one
```

5.2 *Avec des entiers : nombres pairs, nombres impairs*

Exercice 5.2.10 (votrelogin_5210.lean)

1. Donnez, dans L∃∀N , la définition d'un entier impair .
2. Enoncez, puis démontrez dans Lean, le théorème suivant : "La somme de deux entiers naturels pairs est un nombre entier pair"
3. Ecrivez la preuve papier

Vous pouvez utiliser le lemme ci-dessous :

```
#check mul_add
```

Exercice 5.3.11 (votrelogin_5311.lean)

On donne :

```
import data.real.basic

definition fonction := ℝ→ℝ
```

1. Donnez, dans L∃∀N , la définition d'une fonction de ℝ dans ℝ croissante.
2. Enoncez, puis démontrez dans L∃∀N , le théorème suivant : "La composée de deux fonctions croissantes est une fonction croissante".
   (La composée de `f` et de `g` est notée `f ∘g` . Le symbole s'obtient en tapant `\o`).
3. Ecrivez-en une preuve papier.
4. On ajoute la définition suivante de la somme de deux fonctions :

   ```
   def somme_fonctions (f:fonction) (g:fonction):= (λ x:ℝ , f x + g x)
   instance fonction_has_add : has_add fonction :=⟨somme_fonctions⟩
   ```

   La deuxième ligne signifie qu'on pourra écrire `f + g` au lieu de `(somme_fonctions f g)`.
   Enoncez, puis démontrez dans L∃∀N , le théorème suivant : "La somme de deux fonctions croissantes est une fonction croissante".
   Vous pouvez utiliser le lemme ci-dessous :

   ```
   #check add_le_add
   ```

5. Ecrivez-en une preuve papier.

Dans cette section, on va définir dans L∃∀N des notions et prouver des résultats sur les applications. On se donne trois types E, F et G (la notion de type correspond en maths à la notion d'ensemble). Dans L∃∀N , le type `E → F` est le type des fonctions qui à un élément de type E associent un élément de type F.

Au début de votre document, vous définirez ces trois types une fois pour toutes de la façon suivante :

```
variables {E F G:Type}
```

Le mot `variables` signifie que E, F et G seront des paramètres de toutes les fonctions, théorèmes, définitions qui vont suivre, dès lors qu'elles utilisent E, F ou G. Ainsi, si vous définissez la notion d'injectivité pour une application $f : E \to F$, alors, comme E et F seront des paramètres de cette définition, vous pourrez réutiliser cette définition pour exprimer l'injectivité d'une appication $g : F \to G$.

Les accolades (contrairement à des parenthèses) signifient que ces paramètres seront implicites : il sera inutile de les préciser, sauf si L∃∀N ne réussit pas à les deviner.

6.1 *Injectivité, surjectivité*

Exercice 6.1.12 (votrelogin_6112.lean)

Sur le modèle ci-dessous, donnez les définitions d'application injective, surjective, bijective.

```
definition injective  (f: E → F) : Prop  := ∀ (u:E), ∀ (v:E), sorry
```

Exercice 6.1.13 (votrelogin_6113.lean)

Définissez l'application identité de E, puis montrez qu'elle est injective, surjective, bijective.
On rappelle que la flèche à talons des maths ($x : E \mapsto y$) se note `λx:E, y` .
Remarque : exceptionnellement ici, il se peut que L∃∀N ne parvienne pas à résoudre les arguments implicites dans le terme `injective identite`. Il faudra les rendre explicites avec le symbole `@` en écrivant : `@injective E E identite`.

Exercice 6.1.14 (votrelogin_6114.lean)

La composée d'une application `f : E → F` par une application `g : F → G` est l'application `g ∘f` définie par `λx:E, g f x` . Le symbole `∘` s'obtient avec `\o`. Cette opération est nativement définie dans L∃∀N , inutile de la définir.

1. Enoncez puis prouvez un théorème affirmant que la composée de deux applications injectives est injective.
2. Montrez que la composée de deux applications surjectives est surjective.
3. Montrez que la composée de deux applications bijectives est bijective.

6.2 *Image directe, image réciproque*

Exercice 6.2.15 (votrelogin_6215.lean)

Donnez une définition de l'ensemble image d'une application.

Dans L∃∀N , nous avons manipulé la notion de Type, mais il existe également une notion d'ensemble (`set` en anglais). La nuance est subtile. Pour manipuler les ensembles, il vous faudra importer la librairie adéquate, en tapant en première ligne de votre document :

```
import data.set.basic
```

Si `E` est un type, le type `set E` est le type des ensembles d'éléments de type `E`.
Lorsque `x` est de type `E` et que `A` est de type `set E`, on peut utiliser le symbole ∈ dans `x ∈A`.

Exercice 6.2.16 (votrelogin_6216.lean)

Donnez les définitions d'image directe, et d'image réciproque.

La notion d'ensemble ressemble à la notion de type, mais n'est pas identique. Pour désigner l'ensemble des éléments de type F, on ne peut pas utiliser directement `F`, il faut écrire `@set.univ F`.

Exercice 6.2.17 (votrelogin_6217.lean)

Montrez qu'une application $f : E \to F$ est surjective si et seulement si $\operatorname{Im} f = F$.

```
theorem reformulation_surjectivite : ∀  (f: E → F), (surjective f) ↔ (ensemble_image f = @set.univ F) :=
  sorry
```

Vous pourrez avoir besoin de

```
#check set.ext
```

qui permet de justifier l'égalité de deux ensembles A et B à l'aide d'une preuve de $\forall x, x \in A \iff x \in B$.

Exercice 6.2.18 (votrelogin_6218.lean)

Enoncez et prouvez les propriétés suivantes :

1. $\forall f \in F^E, \forall g \in G^F, \forall A \in \mathcal{P}(E), g\langle f\langle A\rangle\rangle = g \circ f\langle A\rangle$
2. $\forall f \in F^E, \forall g \in G^F, \forall C \in \mathcal{P}(G), f^{-1}\left\langle g^{-1}\langle C\rangle\right\rangle = (g \circ f)^{-1}\langle C\rangle$
3. $\forall f \in F^E, \forall A \in \mathcal{P}(E), \forall B \in \mathcal{P}(E), f\langle A \cap B\rangle \subset f\langle A\rangle \cap f\langle B\rangle$
4. $\forall f \in F^E, \forall A \in \mathcal{P}(E), \forall B \in \mathcal{P}(E), f\langle A \cup B\rangle = f\langle A\rangle \cup f\langle B\rangle$
5. $\forall f \in F^E, \forall H \in \mathcal{P}(F), \forall K \in \mathcal{P}(F), f^{-1}\langle H \cup K\rangle = f^{-1}\langle H\rangle \cup f^{-1}\langle K\rangle$
6. $\forall f \in F^E, \forall H \in \mathcal{P}(F), \forall K \in \mathcal{P}(F), f^{-1}\langle H \cap K\rangle = f^{-1}\langle H\rangle \cap f^{-1}\langle K\rangle$

Une construction utile : ***let*** $\cdots$ ***in***

Elle sert à introduire une notation intermédiaire pour alléger un terme de preuve. Syntaxe (parfois le : `type` peut être omis) :

```
let nouveau_symbole : type := expression in
terme_de_preuve
```

Exemple : montrons, comme dans l'ex18 TD7, question 3, que si $u \in F^E$ et $v \in G^F$, et si $v \circ u$ est surjective alors v est surjective.

```
theorem exemple_let_in :  ∀ u:E → F,  ∀ v:F → G, surjective (v ∘  u) → (surjective v) :=
assume (u:E → F) (v:F → G),
  assume h_v_o_u_surjective : surjective (v ∘  u)  ,
    assume (z:G),
      exists.elim (h_v_o_u_surjective z) (
        assume (x:E) (h_z_eq_v_o_u_x : z = (v ∘  u) x),
          let y: F := u x in exists.intro y (
            show z = v y, from h_z_eq_v_o_u_x
          )
      )
```

Une autre construction utile : `eq.subst` *encore noté* ▶

Ce symbole (▶) (obtenu avec `\t`) est une abréviation de `eq.subst` (vous pouvez demander `#check @eq.subst`). Il sert à remplacer une expression par une autre, si elles sont égales, dans un terme. Cette construction constitue une alternative à `congr_arg`.
Syntaxe : si `h_ab` est une preuve de `a=b` et que `h_Pa` est une preuve de `P a` alors `h_ab ▶h_Pa` est une preuve de `P b`. Exemple :

```
theorem exemple_eq_subst : ∀ A:set E, ∀ B:set E, ∀ x:E, x ∈ A ∧ A=B → x ∈ B :=
  assume (A:set E) (B:set E), assume (x:E),
    assume (h: x ∈ A ∧ A=B),
      show x∈ B, from (h.right: A=B) ▶ (h.left: x∈ A )
```

6.3 *Caractérisations de l'injectivité et de la surjectivité*

Exercice 6.3.19 (votrelogin_6319.lean)

Enoncez et prouvez les propriétés suivantes :

1. $\forall f \in F^E, \forall H \in \mathcal{P}(F), f\left\langle f^{-1}\left\langle H\right\rangle\right\rangle \subset H$
2. $\forall f \in F^E, \forall H \in \mathcal{P}(F), f\left\langle f^{-1}\left\langle H\right\rangle\right\rangle = H \cap \operatorname{Im} f$
3. $\forall f \in F^E, \forall A \in \mathcal{P}(E), A \subset f^{-1}\left\langle f\left\langle A\right\rangle\right\rangle$

Exercice 6.3.20 (votrelogin_6320.lean)

Enoncez et prouvez les propriétés suivantes :

1. $\forall f \in F^E$, f surjective $\iff \left(\forall H \in \mathcal{P}(F), f\left\langle f^{-1}\left\langle H\right\rangle\right\rangle = H\right)$
   (Question plus difficile.) Vous pourrez avoir besoin de :

```
#check set.nonempty
#check set.not_nonempty_iff_eq_empty.not_right
#check set.singleton_nonempty
#check set.not_nonempty_iff_eq_empty
#check set.subset_eq_empty
#check set.eq_of_mem_singleton
```

   Il peut être utile de prouver d'abord le lemme ci-dessous :

```
∀  f: E → F, ∀ A: set E,  set.nonempty (image_directe f A) → set.nonempty A
```

2. $\forall f \in F^E$, f injective $\iff \left(\forall A \in \mathcal{P}(E), A = f^{-1}\left\langle f\left\langle A\right\rangle\right\rangle\right)$ Vous pourrez avoir besoin de :

```
#check @set.eq_of_mem_singleton
#check @set.mem_singleton
#check subset_antisymm
#check subset_antisymm_iff
#check subset_refl
```

   Il peut être utile de prouver d'abord le lemme ci-dessous :

```
∀  (f: E → F), ∀ (x:E), (image_directe  f {x}) = {f x}
```

3. $\forall f \in F^E$, f surjective $\iff \left(\forall H \in \mathcal{P}(F), \forall K \in \mathcal{P}(F), f^{-1}\left\langle H\right\rangle \subset f^{-1}\left\langle K\right\rangle \implies H \subset K\right)$
   Vous pourrez avoir besoin de :

```
#check @set.mem_of_mem_of_subset
#check @set.singleton_nonempty
#check @set.nonempty.not_subset_empty
#check @set.not_subset
#check @mt
#check @set.mem_singleton_iff
```

4. $\forall f \in F^E$, f injective $\iff \left(\forall A \in \mathcal{P}(E), \forall B \in \mathcal{P}(E), f\left\langle A \cap B\right\rangle = f\left\langle A\right\rangle \cap f\left\langle B\right\rangle\right)$
5. $\forall f \in F^E$, f bijective $\iff \left(\forall A \in \mathcal{P}(E), f\left\langle \complement_E A\right\rangle = \complement_F f\left\langle A\right\rangle\right)$

6.4 *Un exercice du TD*

Exercice 6.4.21 (votrelogin_6421.lean)

On demande ici de refaire l'exercice 18 du TD7 dans L∃∀N .
Etant donnés $u \in F^E$ et $v \in G^F$, montrez que :

1. Si $v \circ u$ est injective alors u est injective.
2. Si $v \circ u$ est injective et que u est surjective alors v est injective.
3. Si $v \circ u$ est surjective alors v est surjective. (refaites le sans regarder la solution).
4. Si $v \circ u$ est surjective et que v est injective alors u est surjective.

Durant cette séance on va travailler sur la notion de limite d'une suite de nombres réels. La manipulation des nombres réels nécessite l'importation de la librairie idoine (au début du document) :

```
import data.real.basic
```

On définit ensuite la notion de suite de nombres réels de la façon suivante :

```
definition suite := ℕ→ℝ

definition suite_add (u:suite) (v:suite) : suite := λ n:ℕ , ((u n) + (v n) )
instance suite_has_add : has_add suite :=⟨suite_add⟩

definition suite_mul (u:suite) (v:suite) : suite := λ n:ℕ , ((u n) * (v n) )
instance suite_has_mul : has_mul suite :=⟨suite_mul⟩
```

Cela permet de donner un sens aux notations `u+v` et `u*v` qui renvoient respectivement à l'addition et à la multiplication terme à terme de deux suites.

La valeur absolue d'un nombre réel `x` se notera `abs x` ou encore `|x|`. Le symbole `ε` s'obient en tapant `\eps`. Le symbole `ℓ` s'obient en tapant `\ell`.

Exercice 7.0.22 (votrelogin_7022.lean)

1. Donnez la définition d'une suite $(u_n)_{n\in\mathbb{N}}$ convergente vers une limite $\ell \in \mathbb{R}$.

```
definition converge_vers (u:suite) (ℓ : ℝ) : Prop := sorry
```

2. En utilisant la définition précédente, donnez la définition d'une suite convergente .

```
definition converge (u:suite) : Prop := sorry
```

Les théorèmes ci-dessous pourront vous être utiles dans l'exercice suivant. Lequel d'entre eux s'appelle "l'inégalité triangulaire" ?

```
#check @zero_lt_three
#check @mul_two
#check @zero_div
#check @eq_of_sub_eq_zero

#check sub_add_eq_sub_sub
#check sub_eq_add_neg
#check add_assoc
#check add_comm

#check @mul_le_mul_left
#check @mul_le_mul_right
#check @div_le_div_right
#check @div_eq_iff
#check @div_mul_cancel

#check add_le_add
#check @lt_or_eq_of_le
#check @le_max_right
#check @le_max_left

#check @abs_nonneg
#check @abs_add
#check @abs_sub_comm
#check @abs_eq_zero

#check @ge_trans
#check @trans_rel_left
#check @trans_rel_right

theorem not_three_le_two  : ¬ (3:ℝ )≤ 2 := not_le_of_gt (norm_num.lt_bit0_bit1 _ _ (le_refl 1))

theorem lt_div2 (x:ℝ ) (y:ℝ ) (h: x<y) : (x/2) < (y/2) := (div_lt_div_right zero_lt_two).mpr h

theorem deux_moities (u:ℝ ) : (u/2)+(u/2) = u :=
      calc
        (u/2) + (u/2) = (u/2)*1+ (u/2)*1 : (eq.symm (mul_one (u/2))) ▶ (eq.refl ((u/2) + (u/2)))
        ...           = (u/2)*(1+1)      : eq.symm (mul_add _ _ _)
        ...           = (u/2)*2          : congr_arg (λ (z:ℝ ), (u/2)*z) (eq.refl _)
        ...           = u                : div_mul_cancel u two_ne_zero
```

Exercice 7.0.23 (votrelogin_7023.lean)

1. Enoncez, puis démontrez un théorème exprimant que si elle existe, la limite d'une suite est unique.
2. Enoncez, puis démontrez un théorème déterminant la convergence et la limite de la somme de deux suites convergentes.
3. Enoncez, puis démontrez un théorème déterminant la convergence et la limite du produit de deux suites convergentes.

8 ANNEXES

8.1 *Annexe 1 : Fiche : Raccourcis pour les symboles mathématiques*

Symbole	Raccourci
$\forall$	`\all`
$\exists$	`\ex`
$\rightarrow$	`\r`
$\leftrightarrow$	`\lr`
$\neg$	`\not`
$\wedge$	`\and`
$\vee$	`\or`
$\geqslant$	`\ge`
$\circ$	`\o`
λ	`\la`
$\mathbb{N}$	`\N`
$\mathbb{R}$	`\R`
$\langle$	`\<`
$\rangle$	`\>`
$\leqslant$	`\le`
λ	`\la`
$\in$	`\in`
α	`\alp`
$\blacktriangleright$	`\t`
ε	`\eps`
ℓ	`\ell`

8.2 *Annexe 2 : Correspondance approximative de certains mots clés L∃∀N ↔ Maths*

L∃∀N	Maths
$\rightarrow$	$\Longrightarrow$
`assume`	Soit
`assume`	Supposons
`f x`	$f(x)$
`λx, y`	$x \mapsto y$