TP 1 - Prise en main de LEAN

Pour commencer.

1. Connectez-vous au serveur LEAN 4 disponible à l'adresse `https://live.lean-lang.org/`.

2. Vous pouvez taper des commandes dans le bloc de texte de gauche. Les messages de la console s'affichent dans la colonne de droite au fur et à mesure que vous tapez du texte.

*3. Des onglets en haut à droite vous permettront de sauvegarder votre travail sur votre ordinateur, ou bien de charger un travail existant au format *.lean (mais un copier-coller depuis un fichier texte marche tout aussi bien).*

4. On peut écrire des commentaires en LEAN en les précédant de deux tirets :
`-- Ceci est un commentaire`.
N'hésitez pas à écrire des commentaires pour structurer votre code.

1 Types

En LEAN, tous les objets ont un type bien défini. On peut demander à LEAN d'afficher le type d'un objet en tapant la commande `#check`.

1. Tapez les commandes suivantes et expliquez ce qui se produit :

```
#check 2
#check 3.2
#check (-1)
```

Les types suivants seront très utiles dans nos premières séances de TD.

- `Nat` *: type des entiers naturels (les objets de ce type correspondent aux éléments de* $\mathbb{N}$*) ;*
- `Int` *: type des entiers relatifs (les objets de ce type correspondent aux éléments de* $\mathbb{Z}$*) ;*
- `float` *: type des nombres réels* **approchés** *(un objet de ce type correspond à une écriture approchée en binaire d'un nombre réel, avec un nombre fini de chiffres après la virgule).*

On peut importer le module dédié aux nombres réels (nommé `Mathlib.Data.Real.Basic`*) en tapant la commande suivante* au début du fichier *:*

```
import Mathlib.Data.Real.Basic
```

Cela met à disposition le type `Real` *dont les objets correspondent exactement aux éléments de* $\mathbb{R}$*. De même, il existe un module* `Mathlib.Data.Complex.Basic`*, mettant à disposition le type* `Complex`*. Le type* `Rat` *est fourni dans le module* `Batteries.Data.Rat.Basic`.

2. On peut définir des variables en utilisant le mot clé `def`. Essayez les commandes suivantes :

```
def a := 2
def b := 3.2
#check a
#check b
```

Dans les commandes précédentes, le type des variables déclarées est implicite. On peut le rendre explicite avec la syntaxe suivante.

```
def nom_variable : nom_type := valeur
```

3. Essayez les commandes suivantes et expliquer ce qu'elles font :

```
def c : Nat := 2
def d : Int := 2
#check c
#check d
```

Essayez de définir une variable représentant le nombre réel égal à 2.

On peut aussi demander à LEAN de forcer à considérer les expressions numériques comme étant d'un certain type, avec la syntaxe `(nom_objet : nom_type)`*. Par exemple, testez ce qu'affiche le code suivant :*

```
#check 2
#check (2 : Int)
#check (2 : Rat)
```

4. Que se passe-t-il si l'on tape la commande suivante ?

```
def e : Nat := -2
```

LEAN peut aussi évaluer des expressions au moyen de la commande `#eval`*.*

5. Essayez les commandes suivantes, et expliquez ce qui se produit.

```
#check (2 + 2)
#eval (2 + 2)

#check (7 - 12)
#eval (7 - 12)

def var := -32
#check ((var * 12) - 13)
#eval ((var * 12) - 13)
```

Dans la question suivante, on va voir que les types utilisés par LEAN sont eux-mêmes des objets ayant un certain type.

6. Que se passe-t-il si l'on tape les commandes suivantes ?

```
#check Nat
#check Int

#check Type
```

Pour LEAN, le nom `Nat`*, en plus de désigner le type des entiers naturels, est donc un objet de type* `Type`*. Cela nous autorise à écrire par exemple les commandes suivantes :*

```
def nouveau_type : Type :=  Nat
def nouveau_type_2 : Type := Int

#check nouveau_type
```

De la même manière, `Type`*, en plus d'être le type des objets comme* `Int` *et* `Nat`*, est aussi un objet d'un certain type, noté* `Type 1`*.*

7. Quel est le type de l'objet `Type 1` ? Quel est le type de l'objet correspondant à ce type ?

Pour nous, il sera assez commode de penser aux objets de type `Type` *comme représentant des ensembles mathématiques : de la même manière que* $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}\,\mathbb{C}$ *désignent des ensembles, les termes* `Nat`*,* `Int`*,* `Rat`*,* `Real` *et* `Complex` *désignent des* types*, i.e. des objets de type* `Type`*.*

2 Définition de fonctions

Voici une syntaxe standard pour définir une fonction en LEAN :

```
def nom_fonction (nom_entree : type_entree) : type_sortie :=
      (expression_sortie)
```

Par exemple, la fonction suivante prend en entrée un nombre entier naturel `x` *et renvoie l'entier naturel* `x+2` *:*

```
def ajoute_deux (x : Nat) : Nat :=
     x + 2
```

8. Définir une fonction `carre` qui prend en entrée un entier naturel `x` et qui renvoie son carré, puis écrire une fonction `carre_int` qui fait la même chose, mais qui prend en entrée un entier relatif.

9. Utiliser la commande `#eval` pour afficher la valeur de 1239^2. Attention, la syntaxe pour appliquer une fonction `f` à un élément `x` est `f x` et non pas `f(x)`.

10. Quels sont les types des fonctions `carre` et `carre_int` ? On peut afficher ces types d'une façon plus agréable en tapant `#check (carre)` plutôt que `#check carre`.

Comme l'indique la question précédente, une fonction prenant un objet de type `E` *et renvoyant un objet de type* `F` *est donc considérée par* `LEAN` *comme un objet de type* $\texttt{E} \to \texttt{F}$*. En fait, il existe une syntaxe alternative permettant de définir directement une fonction comme un objet de type* $\texttt{E} \to \texttt{F}$*.* La voici :

```
def nom_fonction : type_entree → type_sortie :=
    fun nom_entree => (expression_sortie)
```

On aurait donc pu définir la fonction `ajoute_deux` *de la façon suivante :*

```
def ajoute_deux : Nat → Nat :=
    fun x => x + 2
```

(Pour écrire la flèche →, tapez `\to`*)*

11. Qu'affiche le code suivant ?

```
#check (fun (x : Int) => x + 2)
```

Une expression du type `fun (nom_entree : type_entree) => (expression : type_sortie)` *désigne donc un objet de type* `type_entree → type_sortie`*. En conséquence, il est possible de définir des fonctions qui renvoient des fonctions :*

12. Expliquer ce que fait le code suivant.

```
def somme : Int → (Nat → Int) :=
    fun x => ( fun y => x + y)

#check somme
#check (somme 2)
#eval (somme 2) 3
```

Le type de la fonction `somme` *précédente est à proprement parler* `Int→(Nat→Int)`*. Pour simplifier,* `LEAN` *note ce type de façon abrégée* `Int→Nat→Int`*.*

De même, on pourra écrire `somme 2 3` *plutôt que* `(somme 2) 3`*. On pourra penser à la fonction* `somme` *comme étant une fonction prenant un entier relatif et un entier naturel, et renvoyant un entier relatif. L'écriture suivante est synonyme de celle de la question précédente :*

```
def somme_v2 : Int → Nat → Int :=
    fun x y => x + y

#check (somme_v2 2)
#eval (somme_v2 2 3)
```

13. Ecrire une fonction prenant en entrée trois entiers naturels $a, b, c \in \mathbb{N}$, et renvoyant en sortie l'expression $ab + c$.

14. Ecrire une fonction `plus_deux` prenant en entrée une fonction $f : \mathbb{N} \to \mathbb{N}$, et renvoyant en sortie la fonction $g : x \in \mathbb{N} \to f(x) + 2 \in \mathbb{N}$.

15. Ecrire une fonction `composee` prenant en entrée deux fonctions $f, g : \mathbb{N} \to \mathbb{N}$, et renvoyant en sortie la composée $f \circ g : \mathbb{N} \to \mathbb{N}$.

Dans les exemples précédents, vous pouvez essayer d'appliquer les fonctions `plus_deux` *et* `composee` *à des fonctions de votre choix, puis à des entiers naturels explicites (n'oubliez pas d'utiliser la commande* `#eval` *pour afficher les résultats).*