INF3034L

LC : Logique classique

TP

2025 – 2026

Table des matières

LC TP 1

Programmation fonctionnelle en COQ- GALLINA, premières preuves en logique propositionnelle

Fichier fourni : `lc_tp1.v`

Objectifs :

— Aller plus loin avec le langage de programmation GALLINA,

— Premières preuves en logique propositionnelle.

EXERCICE 1 ▶ **Mise en route**
Téléchargez `lc_tp1.v` et ouvrez le avec COQIDE. Vous pouvez également l'ouvrir avec VISUAL STUDIO CODE si vous avez installé COQ sur votre système et l'extension VSCOQ de VSCODE. Vous écrirez votre code et le compilerez directement dans ce fichier, qui reprend le canevas de ce document.

1.1 Listes d'objets de type nat

On considère ici des listes d'objets de type `nat`.

On peut définir de façon inductive un type `nliste` pour les listes d'objets de type `nat`. Le cas de base est bien sûr la liste vide, l'autre règle de construction applique `cons` à un `nat` et une liste de l'ensemble inductif pour créer un nouvel élément de cet ensemble.

```
Inductive nliste : Type :=
  | vide : nliste
  | cons : nat -> nliste -> nliste.

Definition liste0 := vide.
Definition liste1 := cons 1 vide.
Definition liste2 := cons 2 (cons 1 vide).
Definition liste3 := cons 3 (cons 2 (cons 1 vide)).
Definition liste4 := cons 4 (cons 3 (cons 2 (cons 1 vide))).
Print liste0.
Print liste1.
Print liste2.
```

EXERCICE 2 ▶
Écrire une fonction `ajoute : nat -> nliste -> nliste` telle que `ajoute n l` retourne une liste correspondant à l'ajout de l'élément `n` à la liste `l`. C'est bien sûr juste la fonction qui applique cons.

EXERCICE 3 ▶
Écrire une fonction `longueur` telle que `longueur l` retourne le nombre (`nat`) d'éléments de la liste l. On l'a vue en cours. C'est bien sûr une fonction qui travaille selon la *forme* de `l` : si c'est vide, la longueur vaut zéro, et si `l` est de la forme `cons n l'`, à vous de jouer.

EXERCICE 4 ▶
Écrire une fonction `concat: nliste -> nliste -> nliste` telle que `concat l l'` retourne une liste correspondant à l'ajout des éléments de `l` en tête de la liste `l'`.

EXERCICE 5 ▶
Écrire une fonction `recherche: nat -> nliste -> bool` telle que `recherche n l` retourne `true` si un élément `n` appartient à la liste `l` et `false` sinon.
Pour l'égalité entre éléments du type `nat`, soit on la redéfinit, soit on utilise `Nat.eqb`

```
Require Import Nat.
Check (eqb 3 4).
Compute (eqb 3 4).
```

EXERCICE 6 ▶
Écrire une fonction `miroir: nliste -> nliste`, qui retourne une liste correspondant à son argument dans l'ordre inverse. Dans un premier temps, on pourra utiliser la fonction de concaténation vue précédemment.

EXERCICE 7 ▶ **à faire chez vous**
Écrire une fonction `supprime: nat -> nliste -> nliste` telle que `supprime n l` retourne une liste d'objets de type `nat` correspondant à `l` sans la première occurrence de `n` (le cas échéant), à `l` sinon.

EXERCICE 8 ▶ **à faire chez vous**
Écrire une fonction `supprime_tout: nat -> nliste -> nliste` telle que `supprime_tout n l` retourne une liste correspondant à `l` sans occurrence d'un `nat n` (le cas échéant), à `l` sinon. *)

EXERCICE 9 ▶ **à faire chez vous**
Écrire une fonction `il_existe_pair: nliste -> booleens`, telle que `il_existe_pair l` retourne `Vrai` si un élément de `l` est pair, `Faux` sinon.

EXERCICE 10 ▶ **à faire chez vous**
Écrire dans un premier temps une fonction `leq : nat -> nat -> bool` qui teste si le premier entier est inférieur ou égal au second.
Écrire une fonction `insertion_triee : nat -> nliste -> nliste` qui effectue une insertion triée dans une liste.

EXERCICE 11 ▶ **à faire chez vous**
Écrire une fonction `tri_insertion : nliste -> nliste` qui effectue le tri par insertion d'une liste.

1.2 Premières preuves en logique propositionnelle

Premières tactiques : Mots-clés `assumption`, `intro`, `apply`, `destruct`, `split`, `left` et `right`

La flèche

- Axiome : `assumption`
- Introduction de la flèche : `intro [nom qu'on donne à l'hypothèse]`
- Élimination de la flèche : `apply [nom de l'hypothèse utilisée]`

EXERCICE 12 ▶
Montrer `P -> (P -> Q) -> Q`.

EXERCICE 13 ▶
Montrer `(P -> Q) -> (Q -> R) -> (P -> R)`.

Le et

- Décomposition du `/\` en hypothèse : `destruct [nom de l'hypothèse avec /\]`
- introduction du `/\` : `split`

EXERCICE 14 ▶
Montrer `(P -> Q) /\ (Q -> R) -> (P -> R)`.

EXERCICE 15 ▶
Montrer `P -> Q -> P /\ Q`.

Le ou

- Introduction du \/ :
  - Depuis la droite : `right`
  - Depuis la gauche : `left`
- Decomposition du \/ en hypothèse : `destruct [nom de l'hypothèse avec \/]`

EXERCICE 16 ▶
Montrer `(P \/ Q) -> (Q \/ P)`.

Ex falso

`destruct` donne un sous but par constructeur.
Comme `False` n'a aucun constructeur : `destruct` résout le but.

EXERCICE 17 ▶
Montrer `False -> P`.

- Remplacer tout but par `False` : `exfalso`

EXERCICE 18 ▶
Montrer `(P -> False) -> P -> (Q \/ (R -> S /\ T) -> U)`.

LC TP 2

Programmation fonctionnelle en COQ- GALLINA, encore des preuves en logique propositionnelle

Fichier fourni : `lc_tp2.v`

Objectifs :

— Écrire des fonctions sur les listes qui seront utilisées dans les TP suivants,

— Encore quelques preuves en logique propositionnelle.

2.1 Retour sur les listes

On définit le type inductif `nlist` des listes de nat.

```
Inductive nlist : Set :=
| nnil : nlist
| ncons : nat -> nlist -> nlist.
```

On ajoute des notations confortables.

```
Infix "::" := ncons.
Notation "[]" := nnil.
```

EXERCICE 1 ▶
Définir la fonction `concat` qui concatène deux `nlist` `l1` et `l2` (par récursion sur `l1`).

```
Fixpoint concat (l1 l2 : nlist) : nlist :=
[].
```

On note ++ en notation infix pour la concatenation.

```
Infix "++" := concat.
```

EXERCICE 2 ▶
Définir la fonction `length` qui retourne la longueur d'une `nlist`.

EXERCICE 3 ▶
Définir la fonction `appartient` qui retourne `vrai` si un `nat` appartient à une `nlist`.

2.2 Modélisations et preuves en logique propositionnelle

EXERCICE 4 ▸
Modéliser l'exercice de TD *Georges aime la logique*, prouver que Georges aime la logique.

Le ou - suite

— Introduction du `/\` : `split`

EXERCICE 5 ▸
Modéliser l'exercice de TD *Zoé va à Paris*, prouver que Zoé va à Paris.

Le non

— La notation `not` : `unfold not`

— La notation `not` en hypothèse : `unfold not [nom de l'hypothèse avec ~]`

EXERCICE 6 ▸
Montrer `(~P \/ ~Q) -> ~(P /\ Q)`.

— Si on a `toto` et `~toto` dans les hypothèses : le but est alors résolu par `contradiction`

EXERCICE 7 ▸
Montrer `P -> ~P -> Q`.

Le tiers-exclu On introduit la règle de tiers-exclu.

```
Context (Tiers_exclu: forall X: Prop, X \/ ~X).
```

— Pour utiliser le tiers-exclu, c'est-à-dire pour avoir deux sous buts, un avec `toto` en hypothèse, l'autre avec `~toto` : `destruct (Tiers_exclu toto)`.

EXERCICE 8 ▸
Montrer `P \/ ~P`.

EXERCICE 9 ▸
Montrer `((P -> Q) -> P) -> P`.

EXERCICE 10 ▸
Modéliser l'exercice de TD *Frodon est triste*, prouver que Frodon est triste.

EXERCICE 11 ▸
Modéliser l'exercice de TD *Gérard est déprimé*, prouver que Gérard est déprimé.

EXERCICE 12 ▸
Montrer `(~~P -> P) /\ (P -> ~~P)`.
Pour l'un des deux sens on aura besoin du tiers-exclu et, en remarquant qu'on peut déduire `False` des hypothèses, de la simplification `exfalso`.

2.3 Encore des preuves en logique propositionnelle

EXERCICE 13 ▸
Montrer `(P \/ Q) -> (Q \/ P)`.

EXERCICE 14 ▸
Montrer `(P -> Q -> R) <-> (P /\ Q -> R)`.

EXERCICE 15 ▸
Montrer `(P -> Q) /\ (P -> R) <-> (P -> Q /\ R)`.

LC TP 3
Fonctions non récursives sur les types inductifs

Fichier fourni : `lc_tp3.v`

3.1 Le prédicat = et le quantificateur universel

Un prédicat = est déjà défini en Coq. On peut considérer qu'il s'agit de la plus petite relation réflexive.
C'est un inductif avec une seule règle de construction : pour tout x on construit x=x.
La règle d'introduction de = est `reflexivity`

EXERCICE 1 ▶
Montrez 4=4.

La règle d'élimination de = est `rewrite`. Lorsqu'on a une ÉGALITÉ `x = y` dans l'hypothèse `Heq`,

— On peut remplacer dans le but tous les x libres par des y avec `rewrite -> Heq`
— On peut remplacer dans le but ous les y libres par des x avec `rewrite <- Heq`
— On peut remplacer dans une hypothèse H tous les x libres par des y avec `rewrite -> Heq in H`
— On peut remplacer dans une hypothèse H tous les y libres par des x avec `rewrite <- Heq in H`

EXERCICE 2 ▶
Montrez `(x : nat) : 1 + (x + 3) = 6 -> 1 + (x + 3) = 1 + x + 3  -> 1 + (1 + x + 3) = 1 + 6.`

En Coq des CONSTRUCTEURS DIFFÉRENTS donnent des TERMES DIFFÉRENTS.
Si en hypothèse on trouve le prédicat d'égalité avec deux membres différents alors on peut achever la preuve directement avec `discriminate`

EXERCICE 3 ▶
Montrez `3=4 -> False`.

La tactique utilisée pour la règle d'introduction de l'universel est `intro nom_de_la_variable_générique`
On peut être amené à faire des calculs : `simpl` dans le but ou `simpl in Hp` dans l'hypothèse Hp,
ou encore `cbv` dans le but ou `cbv in Hp` dans l'hypothèse Hp.

EXERCICE 4 ▶
Montrez `forall (x : nat), 2 + 3 + x = 5 + x.`

On reprend les booléens du TP précédent.

```
Inductive booleens : Type :=
| Vrai : booleens
| Faux : booleens.
Definition ou (a : booleens) (b : booleens) : booleens :=
match a with | Vrai => Vrai | Faux => b end.
Definition et (a : booleens) (b : booleens) : booleens :=
match a with | Vrai => b | Faux => Faux end.
```

EXERCICE 5 ▶
Montrez que Faux ou un booléen c'est ce booléen, et que Faux et un booléen c'est Faux.

3.2 Fonctions non-récursives sur les types inductifs

Dans la suite on va utiliser les tactiques vues jusqu'ici :

— intro [Id1]
— destruct [Hypothèse]
— split
— left
— right
— discriminate
— reflexivity
— simpl (in [Hypothèse])
— cbv (in [Hypothèse])
— apply [Théorème] (in [Hypothèse])
— rewrite [Théorème d'égalité] (in [Hypothèse])

3.2.1 Symboles

On définit un petit alphabet (les symboles) d'exemple : c'est juste une énumération, représentée en Coq par un type inductif avec 2 constructeurs sans argument (des constantes).

```
Inductive Symbole : Type :=
| a : Symbole
| b : Symbole.
```

Ici, `Symbole` est le plus petit ensemble qui contient `a`, `b` et rien d'autre, donc intuitivement, `Symbole` est l'ensemble `{a,b}`.

EXERCICE 6 ▶
Définissez une fonction `comp_symboles` qui teste si deux symboles sont égaux.

EXERCICE 7 ▶
On va montrer que `(comp_symboles x y) = true` si et seulement si `(x = y)`. Autrement dit, `comp_symbole` *décide* l'égalité entre deux éléments de `Symbole`.

On peut décomposer la preuve :

— Prouvez `comp_symboles_correct` : si `comp_symboles x y = true` alors `x = y`,
— Prouvez `comp_symboles_complet` : si `x = y` alors `comp_symboles x y = true`.

EXERCICE 8 ▶
Enoncez et prouvez la propriété que comparer un `Symbole` avec lui-même renvoie vrai.

HINT : `com_symboles_complet` fait exactement ce dont on a besoin, on peut donc l'utiliser.

EXERCICE 9 ▶
Montrez `forall (x : Symbole), x = a \/ x = b`.

HINT : `destruct x`, `left` ou `right` pour `\/`.

3.2.2 option nat

EXERCICE 10 ▶
Définissez la fonction `comp_option_nat` qui renvoie `true` si les deux `option nat` sont égaux.

Par convention, si les deux `option nat` valent `None`, la fonction retourne `true`.

EXERCICE 11 ▶
Enoncez et prouvez la propriété que la fonction `comp_option_nat` est correcte et complète.

HINT : si la tactique `cbv` ne calcule pas *assez*, remplacez le nom de fonction par sa valeur avec `unfold`, comme par exemple avec `unfold not` qui remplace `~A` par `A -> False`.

```
Check PeanoNat.Nat.eqb_eq.
```

Pour la complétion on pourra se servir du lemme suivant :

```
Lemma some_injective : forall (n:nat) (n':nat), Some n = Some n' -> n = n'.
Proof.
intro n.
intro n'.
intro Heq.
inversion Heq. reflexivity.
(* ou injection Heq as Heq'. assumption. *)
Qed.
```

3.2.3 Paires d'entiers

EXERCICE 12 ▸
Montrez `forall (p : nat*nat), p = (fst p, snd p).`

EXERCICE 13 ▸
Prouvez que `swap` est involutive.
Rappel, une fonction `f` est une involution ssi quel que soit `x`, `f(f(x)) = x`.
HINT : pour une paire `p`, utiliser `destruct p` pour retrouver `(a,b)`.

```
Definition swap (p : nat * nat) : nat * nat :=
match p with
 | (x,y) => (y,x)
end.
```

EXERCICE 14 ▸
Enoncez et prouvez la propriété que la fonction `comp_pair_nat` est correcte et complète.
HINT : utiliser `Bool.andb_true_iff`.

```
Bool.andb_true_iff :
    forall b1 b2 : bool, (b1 && b2)%bool = true <-> b1 = true /\ b2 = true
```

Ce théorème lie le ET des booléens, la fonction `andb` en Coq (qui se note aussi `&&`) et le ET logique, noté `/\` en Coq (qui se note aussi `and`).

```
Check Bool.andb_true_iff.
Check PeanoNat.Nat.eqb_eq.
```

```
Definition comp_pair_nat (p q : nat * nat) : bool :=
match p with
| (xp,yp) => match q with
             | (xq,yq) => andb (Nat.eqb xp xq) (Nat.eqb yp yq)
             end
end.
```

LC TP 4
Fonctions récursives sur les types inductifs

Fichier fourni : `lc_tp4.v`

4.1 Le quantificateur universel

Pour écrire une quantification existentielle, on utilise le mot-clé `exists`.
Par exemple :

```
forall (l : nlist), length l <> 0 -> exists (n : nat), exists (l' : nlist), l = n::l'
```

La tactique utilisée pour la règle d'introduction de l'existentiel est `exists un_terme_particulier`.

EXERCICE 1 ▸
Montrez qu'il existe un `x` tel que la fonction `plus` appliquée à ce `x` et à `0` retourne `0`.

EXERCICE 2 ▸
Montrez que pour tout `x`, il existe un /verb!y! tel que la fonction appliquée à `x` et `y` retourne `x + 1`.

4.2 Fonctions récursives et induction sur les entiers

On rappelle que les objets de type `nat` sont définis inductivement de façon similaire à

```
Inductive entiers : Set :=
   | Z : entiers
   | Succ : entiers -> entiers. *)
```

On dispose donc d'un principe d'induction `nat_ind`, construit à peu près comme vu en cours.

```
forall P : nat -> Prop,
     P 0
  -> (forall (n' : nat), P n' -> P (S n'))
                   -> forall (n : nat), P n
```

Si on omet le `forall P` qui n'est pas du premier ordre, on se retrouve bien avec deux branches :

— Une branche qui demande de prouver sur le cas de base des `nat`, c'est-à-dire `0`,

— Une branche qui demande de prouver sur un `nat` construit par `S` à partir d'un `nat` sur lequel on sait déjà prouver la propriété.

On peut en déduire la propriété sur tout `nat` obtenu par `0` et `S`.

En Coq, l'application de la tactique `induction` sur un nom d'entier produira donc **deux** sous-buts (il y a bien 2 règles de construction des entiers) :

— Le sous-but correspondant au cas de base O,

— Le sous-but correspondant au cas inductif où l'hypothèse d'induction apparaît dans le contexte.

Comme on sait que ça va mettre deux nouvelles choses dans la branche de droite et rien de nouveau dans celle de gauche, on peut nommer directement : `induction "n" as [ | "m" "Hyp_Ind_m"]`, où `n` est dans le cas de droite l'entier `Succ m` avec comme hypothèse d'induction que la propriété est vraie pour `m` (hypothèse nommée ici `Hyp_Ind_m`).

EXERCICE 3 ▸
Montrez que la fonction plus appliquée à `0` et à un `y` quelconque retourne ce `y`.

La définition de `plus` est récursive sur le paramètre de gauche, donc pas de problème ici, c'est juste un calcul (`simpl`).

EXERCICE 4 ▶
Montrez que la fonction plus appliquée un `x` quelconque et `0` retourne ce `x`.
Là il faut travailler par induction sur `x`. On utilise `induction x as...` qui invoque la règle `nat_ind`.

EXERCICE 5 ▶
Montrez que le successeur du résultat de la fonction `plus` appliquée à `x` et `y` quelconques, c'est la fonction `plus` appliquée au successeur de `x` et à `y`.
Comme avant, comme `plus` est récursive à gauche, c'est juste un calcul.

EXERCICE 6 ▶
Montrez que le successeur du résultat de la fonction `plus` appliquée à `x` et `y` quelconques, c'est la fonction `plus` appliquée à `x` et au successeur de `y`.
Pareil qu'avant, on procède par induction sur `x`.

On peut ajouter une hypothèse provenant d'une propriété extérieure à la preuve courante, **si on en a besoin**.
On **spécialize** une propriété :
`specialize propriété with (univ1:=spé1) (univ2:=spé2) ... as Hspé_propriété`
où `univ1` est le premier universel instancié par la valeur spécialisée `spé1` ... et a pour effet de créer une nouvelle hypothèse `Hspé_propriété`, qui est donc la spécialisation de `propriété`.
Par exemple, si on a un `y` en hypothèse, `specialize plus_Z_r with (x:=y) as Hspe_plus_Z_r` crée l'hypothèse `Hspe_plus_Z_r : y + 0 = y`.

EXERCICE 7 ▶
Montrez que la fonction plus est commutative.
On pourra utiliser les propriétés montrées ci-dessus.

4.3 Fonctions récursives et induction sur les listes d'entiers

On reprend les listes de `nat` du TP précédent :

```
Inductive nlist : Set :=
| nnil : nlist
| ncons : nat -> nlist -> nlist.
```

Avec des notations confortables :

```
Infix "::" := ncons.
Notation "[]" := nnil.
```

On dispose donc d'un principe d'induction `nlist_ind`, similaire à `nat_ind` :

```
forall P : nlist -> Prop,
       P []
    -> (forall (n : nat) (l' : nlist), P l' -> P (n :: l'))
                    -> forall (l : nlist), P l
```

Si on omet le `forall P` qui n'est pas du premier ordre, on se retrouve bien avec deux branches :

— Une branche qui demande de prouver sur le cas de base des listes,

— Une branche qui demande de prouver sur une liste construite par `::` à partir d'une liste sur laquelle on sait déjà prouver la propriété.

On peut en déduire la propriété sur toute liste obtenue par `[]` et `::`.
Comme pour les entiers, on sait qu'il n'y a rien dans la branche de gauche (cas de base) et qu'il y aura **trois** nouvelles choses dans la branche de droite, on les nomme directement :
`induction "l" as [ | "n" "l'" "IHl'"]`,
où `l` est dans le cas de droite la liste `n::l'` (`n` est un élément quelconque qui ne nous intéresse pas) avec comme hypothèse d'induction que la propriété est vraie pour `l'`, hypothèse nommée ici `IHl'`.

```
Fixpoint concat (l1 l2 : nlist) : nlist :=
  match l1 with
  | []     => l2
  | x :: l => x::(concat l l2)
  end.

(* On note ++ en notation infix pour la concatenation *)
Infix "++" := concat.

Fixpoint length (l : nlist) : nat :=
  match l with
  | []     => 0
  | x :: l => S(length l)
  end.

Fixpoint appartient (x : nat) (l : nlist) : bool :=
match l with
| [] => false
| h::rl => (Nat.eqb x h) || (appartient x rl)
end.
```

EXERCICE 8 ▶
Montrez que la fonction `length` retourne 0 **seulement si** la liste est vide.

EXERCICE 9 ▶
Prouvez que pour tout `nat x` et toute `nlist l`, la liste vide n'est pas obtenue par l'ajout de `x` en tête de `l`.

EXERCICE 10 ▶
Exprimez et montrez que pour tout élément `x` et toutes listes `l1` et `l2`, ajouter `x` en tête de la concaténation de `l1` et `l2` est la même chose que concaténer `l1` avec `x` en tête et `l2`.

EXERCICE 11 ▶
Exprimez et montrez que la fonction `length` appliquée à la concaténation de deux listes quelconques `l1` et `l2` retourne la somme des applications de cette fonction à chacune des deux listes.

EXERCICE 12 ▶
Exprimez et montrez que pour toute liste `l2`, concaténer la liste vide à `l2` renvoie exactement la liste `l2`.

EXERCICE 13 ▶
Exprimez et montrez que pour toute liste `l1`, concaténer `l1` à la liste vide renvoie exactement la liste `l1`.

EXERCICE 14 ▶
Exprimez et prouvez la propriété que l'appartenance d'un élément à une liste vide est fausse.

EXERCICE 15 ▶
Exprimez et prouvez la propriété que l'appartenance d'un élément à une liste singleton est vraie ssi l'élément recherché et celui du singleton sont égaux.

EXERCICE 16 ▶
Montrez que la fonction `appartient` est correcte et complète.

On exprimera que `(appartient x l)` est vrai **si et seulement si** il existe une décomposition de `l` de la forme `l = l1 ++ x :: l2`.