TP 1

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

Fichier fourni : `lc_tp1.v`

Objectifs :

- Se familiariser avec l'environnement COQIDE et le langage de programmation GALLINA,
- Se familiariser avec 4 mots-clés : `Definition`, `Definition` avec `match ... with`, `Inductive`, `Fixpoint`,
- 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.

Définir un objet (entier, fonction ...) : Mot-clé `Definition`
`Definition` <nom de l'objet> : <type de l'objet> := <valeur de l'objet> .

```
Definition a : nat := 3.
Definition b : nat := 6.
```

Effectuer un calcul dans l'interpréteur : directive `Compute`

```
Compute (a+b).
```

Afficher le type dans l'interpréteur : directive `Check`

```
Check (a+b).
```

Afficher la valeur dans l'interpréteur : directive `Print`

```
Print a.
```

1.1 Types énumérés et inductifs

Définition d'un ensemble inductif : Mots-clés `Inductive` et | (pipe) par cas
On donne des règles. Comme on définit un *type* de données, son propre type est `Type`.

```
Inductive jour : Type :=
  | lundi : jour
  | mardi : jour
  | mercredi : jour
  | jeudi : jour
  | vendredi: jour
  | samedi : jour
  | dimanche : jour.
```

Définition d'une fonction : Mots-clés `Definition`, `match`, `with` et `end`

Réalisée suivant *la forme* du paramètre, c'est du *filtrage de motif* ou *pattern matching*. C'est le mécanisme le plus confortable pour manipuler des structures inductives.

```
Definition jour_suivant (j : jour) : jour :=
  match j with
  | lundi => mardi
  | mardi => mercredi
  | mercredi => jeudi
  | jeudi => vendredi
  | vendredi => samedi
  | samedi => dimanche
  | dimanche => lundi
  end.
```

EXERCICE 2 ▶
Définir la fonction qui retourne le surlendemain d'un jour donné. C'est une fonction qui **appliquée à** un jour, **retourne** un jour.

Les booléens

```
Inductive booleens : Type :=
| Vrai : booleens
| Faux : booleens.

Definition non (a : booleens) : booleens :=
  match a with
  | Vrai => Faux
  | Faux => Vrai
  end.
```

EXERCICE 3 ▶
Définir la fonction *et* sur les booléens.

EXERCICE 4 ▶
Définir la fonction *ou* sur les booléens.

EXERCICE 5 ▶ **à faire chez vous**
Définir une fonction `bcompose : f -> g -> h` telle que `h` est la composition des deux fonctions booléennes `f` et `g`.
Tester `bcompose` en définissant une fonction `nonnon : booléens -> booléens` qui définit `non o non`.

Le langage de Coq a bien sûr des booléens (dans le type prédéfini bool), ils sont en fait définis de la même façon que nos booléens. Pour l'instant nous allons continuer de travailler avec les nôtres.

Les entiers

On définit maintenant de façon inductive le type des entiers naturels. Un entier naturel est :

— Soit un élément particulier noté Z (pour zéro, c'est un cas de base ici),

— Soit le successeur d'un entier naturel.

On a deux constructeurs pour les entiers : ils sont soit de la *forme* "Z" soit de la *forme* "Succ d'un entier".

```
Inductive entiers : Type :=
| Z : entiers
| Succ : entiers -> entiers.

Definition un  := Succ Z.
Definition deux  := Succ un.
```

EXERCICE 6 ▶
Définir la fonction prédécesseur `pred`. C'est une fonction qui **appliquée à** un entier, **retourne** un entier.

Définition d'une fonction récursive : Mot-clé `Fixpoint`

On veut écrire une fonction récursive pour ajouter deux entiers. Comme la fonction est récursive, on utilise le mot-clé `Fixpoint` (et non plus `Definition`). Elle se calcule selon la forme du premier paramètre.

```
Fixpoint plus (a : entiers) (b : entiers) : entiers :=
  match a with
  | Z => b
  | Succ n => Succ (plus n b)
  end.
```

EXERCICE 7 ▶
Définir la fonction `mult` qui calcule le produit de deux entiers. Elle se calcule selon la forme du premier paramètre.

EXERCICE 8 ▶
Définir une fonction `est_pair`, telle que `est_pair` appliquée un entier `a` retourne `Vrai` si `a` est pair, `Faux` sinon.

EXERCICE 9 ▶ **à faire chez vous**
Définir la fonction `factorielle` sur les `entiers`.

EXERCICE 10 ▶ **à faire chez vous**
Définir la fonction `moins`, soustraction non négative sur les `entiers`.

EXERCICE 11 ▶ **à faire chez vous**
Définir une fonction `inf`, tel que `inf a b` vaut / retourne `Vrai` si `a` est inférieur ou égal à `b`, `Faux` sinon.

EXERCICE 12 ▶ **à faire chez vous**
Définir une fonction `egal`, tel que `egal a b` donne `Vrai` si les entiers `a` et `b` sont égaux, `Faux` sinon.

Types prédéfinis

Précédemment, on a défini nos booléens et nos entiers naturels, mais ils sont en fait déjà définis dans la bibliothèque que COQ charge initialement au démarrage :

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

avec les fonctions `negb` (complémentaire), `andb` (et, min), `orb` (ou, max).

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

avec les fonctions usuelles `+` , `-` , `*` , etc. et les comparaisons : `Nat.eqb` pour le test d'égalité, `Nat.ltb` pour le test plus petit, `Nat.leb` pour le test plus petit ou égal.

Pour la suite, lorsque cela n'est pas précisé, nous utiliserons les booléens et entiers prédéfinis.

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 13 ▶
Montrer `P -> (P -> Q) -> Q`.

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

Le et

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

— introduction du `/\` : `split`

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

EXERCICE 16 ▶
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 17 ▶
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 18 ▶
Montrer `False -> P`.

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

EXERCICE 19 ▶
Montrer `(P -> False) -> P -> (Q \/ (R -> S /\ T) -> U)`.