TP 1

Programmation fonctionnelle en Coq- Gallina

Fichier fourni : `lf_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`.

Exercice 1 ▶ **Mise en route**
Téléchargez `lf_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 bien 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.
Definition trois  := Succ deux.
```

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 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 13 ▶
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 14 ▶
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 15 ▶
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 16 ▶
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 17 ▶ **à faire chez vous**
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 18 ▶ **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 19 ▶ **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 20 ▶ **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 21 ▶ **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 22 ▶ **à faire chez vous**
Écrire une fonction `tri_insertion : nliste -> nliste` qui effectue le tri par insertion d'une liste.

1.3 Arbres binaires

EXERCICE 23 ▸
Donner une définition par induction de l'ensemble `nBin` des arbres binaires contenant des `nat`.
Deux constructeurs :

- `nEmpty` : arbre vide,
- `nNode` : création d'un noeud avec un fils gauche, un `nat` et un fils droit.

Exemple d'arbre à 5 éléments :

```
Definition a1 := nNode
                      (nNode nEmpty 2 nEmpty)
                      1
                      (nNode
                            (nNode nEmpty 4 nEmpty)
                            3
                            (nNode nEmpty 5 nEmpty)
                      ).

Check a1.
Print a1.
```

EXERCICE 24 ▸
Définir la fonction `nelements` qui renvoie la liste des éléments contenus dans un arbre binaire de `nat`. Le faire naïvement avec un `concat` pour commencer.

EXERCICE 25 ▸ **à faire chez vous**
Définir la fonction `nnelts` qui renvoie le nombre de noeuds internes (portant une étiquette de type `nat`) dans un `nBin`.

EXERCICE 26 ▸ **à faire chez vous**
Définir la fonction `nfeuilles` qui renvoie le nombre de feuilles d'un `nBin`.

EXERCICE 27 ▸ **à faire chez vous**
Définir la fonction `nsum` qui renvoie la somme des valeurs portées par les noeuds internes d'un `nBin`.