Leçon 5 : Induction et Programmation Fonctionnelle

Feuille d'exercices

Aujourd'hui on a vu que le mécanisme `Inductive` s'applique aussi aux **données** : les entiers naturels (`nat`) et les listes (`list`).

Un type inductif est défini par ses **constructeurs** :

— `nat` a deux constructeurs : `O` (zéro) et `S` (successeur).
— `list A` a deux constructeurs : `nil` (liste vide) et `cons` (ajout en tête).

Pour calculer avec ces données, on utilise le filtrage de motifs (`match ... with`). Pour des fonctions qui s'appellent elles-mêmes, on utilise `Fixpoint` . Pour prouver des propriétés, on utilise la tactique `induction` .

Ces exercices vous guident à travers chaque étape. Remplacez `todo` par votre solution. Le préambule du fichier est :

```
Require Import List.
Import ListNotations.
Axiom todo : forall {X : Type}, X.
```

1. Fonctions sur les entiers naturels

Rappel — définition de `nat` .

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

Les notations `0` , `1` , `2` ... sont du sucre syntaxique pour `O` , `S O` , `S (S O)` ... Le `+` standard de Rocq est défini exactement comme `add` vu en cours.

Exercice 1 — la fonction « est-ce zéro ? »

`is_zero n` renvoie `true` si $n = 0$, `false` sinon.

Remarque : `bool` est lui aussi un type inductif, avec deux constructeurs : `true` et `false` .

```
Definition is_zero (n : nat) : bool :=
  match n with
  | O   => todo   (* que renvoie-t-on quand n = 0 ?   *)
  | S p => todo   (* que renvoie-t-on quand n = S p ? *)
  end.

Compute is_zero 0.   (* attendu : true  *)
Compute is_zero 3.   (* attendu : false *)
```

Exercice 2 — doubler un entier

`double n` calcule $n + n$.

Rappel du schéma de récursion sur `nat` :

- Cas de base : `double O = ?`
- Cas récursif : `double (S m) = ?` en utilisant `double m`.

Indice : on sait mathématiquement (par distributivité) que

$$2 \times (m + 1) = (2 \times m) + 2.$$

Il suffit alors d'orienter l'équation pour en faire une règle de calcul dans le cas récursif.

```
Fixpoint double (n : nat) : nat :=
  match n with
  | O   => todo        (* double de 0 ? *)
  | S m => todo        (* double de S m, connaissant [double m] ? *)
  end.

Compute double 0.    (* attendu : 0 *)
Compute double 3.    (* attendu : 6 *)
```

Exercice 3 — la multiplication

`mul n m` calcule $n \times m$.

Indice : multiplier par un successeur, c'est ajouter une fois de plus :

$$(p + 1) \times m = (p \times m) + m.$$

(Vous pourrez utiliser le `+` standard de Rocq.)

```
Fixpoint mul (n m : nat) : nat :=
  (* Comme pour l'addition, vous procèderez par récursion structurelle sur
     le premier argument [n] à l'aide de [match n with ...] *)
  todo.

Compute mul 3 4.    (* attendu : 12 *)
Compute mul 0 5.    (* attendu : 0  *)
```

2. Fonctions sur les listes

Rappel — définition de `list`.

```
Inductive list (A : Type) : Type :=
  | nil  : list A
  | cons : A -> list A -> list A.
```

Notations : `[]` pour `nil`, `x :: l` pour `cons x l`.

`h :: t` se lit : « liste dont la tête est `h` et la queue est `t` ».
La récursion suit la même structure : cas `[]` (liste vide) et cas `h :: t`.

Exercice 4 — longueur d'une liste

`my_length l` renvoie le nombre d'éléments de `l`.

Indice :

— La liste vide a longueur 0.
— La liste `h :: t` a la longueur de sa queue `t` plus 1.

```
Fixpoint my_length {A : Type} (l : list A) : nat :=
  match l with
  | []     => todo   (* longueur de la liste vide ?                    *)
  | h :: t => todo   (* longueur de [h :: t], connaissant [my_length t] ? *)
  end.

Compute my_length [1; 2; 3].        (* attendu : 3 *)
Compute my_length ([] : list nat).  (* attendu : 0 *)
```

Exercice 5 — inverser une liste

`my_rev l` renvoie `l` dans l'ordre inverse.

Indice : pour inverser `h :: t`, on inverse d'abord `t`, puis on place `h` à la suite du résultat avec l'opérateur de concaténation `++`.

Par exemple, inverser `[1; 4; 7]` revient à inverser `[4; 7]`, ce qui résulte en `[7; 4]`, puis à concaténer à droite la liste avec pour unique élément 1 :

`[7; 4]` ++ `[1]` = `[7; 4; 1]`

```
Fixpoint my_rev {A : Type} (l : list A) : list A :=
  (* Vous procèderez par récursion structurelle sur la liste [l] à l'aide
     de [match l with ...] *)
  todo.

Compute my_rev [1; 2; 3].        (* attendu : [3; 2; 1] *)
Compute my_rev ([] : list nat).  (* attendu : []        *)
```

3. Preuves par induction

On prouve maintenant des propriétés sur le `+` standard de Rocq. La clé : `n + m` se calcule (est défini) par récursion sur `n`.

Cela signifie que :

1. `0 + m` se réduit immédiatement en `m` ⇒ `reflexivity` suffit.

2. `S p + m` se réduit en `S (p + m)` ⇒ `simpl` fait cette étape.
3. $n + 0 = n$ pour n quelconque ⇒ il faut `induction`.

On a donc une égalité *définitionnelle* dans les cas 1 et 2, qui peut être résolue automatiquement par Rocq, mais une égalité strictement *propositionnelle* dans le cas 3, qui nécessite une preuve manuelle (par induction).

Schéma général d'une preuve par induction sur `n` :

```
induction n as [| p IH].
- (* but : propriété pour 0 *) ...
- (* but : propriété pour S p, IH donne la propriété pour p
     (hypothèse d'induction) *) ...
```

Exercice 6 — le successeur commute avec l'addition (à droite)

Énoncé : $\forall n, m \in \mathbb{N}, \quad n + (\mathtt{S}\ m) = \mathtt{S}\ (n + m)$

Ce lemme est utile pour la suite — il exprime que « additionner le successeur de m » = « prendre le successeur de $n + m$ ».

Preuve guidée :

— Induction sur `n`.
— Cas `O` : `0 + S m = S (0 + m)`. Les deux côtés se réduisent à `S m`.
— Cas `S p` : but `S p + S m = S (S p + m)`.
Après `simpl`, le but devient `S (p + S m) = S (S (p + m))`.
Utiliser l'hypothèse d'induction `IH : p + S m = S (p + m)`.
Puis `reflexivity`.

```
Lemma add_succ_r : forall n m : nat, n + S m = S (n + m).
Proof.
  intros n m.
  induction n as [| p IH].
  - (* but : 0 + S m = S (0 + m) *)
    admit.
  - (* but : S p + S m = S (S p + m) *)
    (* IH  : p + S m = S (p + m)    *)
    simpl.       (* S p + S m devient S (p + S m), et S p + m devient S (p + m)
    ↪  *)
    admit.       (* utiliser rewrite IH, puis reflexivity *)
Admitted.
```

Exercice 7 — l'addition est associative

Énoncé : $\forall n, m, p \in \mathbb{N}, \quad (n + m) + p = n + (m + p)$

Indice : la structure de la preuve est identique à celle de l'exercice 6, par induction sur `n`.

```
Lemma add_assoc : forall n m p : nat, (n + m) + p = n + (m + p).
Proof.
Admitted.
```

Bonus — longueur de la concaténation

Énoncé :

$$\forall\, A,\ \forall\, \ell_1, \ell_2 : \text{list } A, \quad |\ell_1 \texttt{ ++ } \ell_2| \;=\; |\ell_1| + |\ell_2|$$

Indice : induction sur `l1`. Même schéma que les preuves précédentes, mais sur les listes : cas de base `nil` et cas récursif `cons h t`.

```
Lemma length_app : forall (A : Type) (l1 l2 : list A),
    my_length (l1 ++ l2) = my_length l1 + my_length l2.
Proof.
  intros A l1 l2.
  induction l1 as [| h t IH].
  - admit.
  - admit.
Admitted.
```