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`.