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.
```