TP 2
Programmation fonctionnelle et automates en Coq- Gallina (partie 1)

Fichier fourni : `lf_tp2.v`

Objectifs : Définir tout ce dont on a besoin pour définir des automates et de les faire s'exécuter dans la partie "programme" de Coq :

— Le type `Alphabet` avec une fonction qui teste l'égalité,

— Le type prédéfini `option` pour représenter les fonctions partielles,

— Le type prédéfini `prod A B` des paires,

— La recherche dans une liste d'entiers et dans une liste de paires.

2.1 L'alphabet et son égalité calculable

On définit un petit alphabet d'exemple : c'est juste une énumération, représentée en Coq par un type inductif avec 2 constructeurs sans argument (des constantes).

```
Inductive Alphabet : Type :=
| a : Alphabet
| b : Alphabet.
```

Ici, `Alphabet` est *le plus petit ensemble qui contient a, b et rien d'autre*, donc intuitivement, `Alphabet` est l'ensemble `{a,b}`.

Exercice 1 ▸ **Égalité de deux éléments de l'alphabet**
Définir une fonction `comp_alphabet` qui teste si deux éléments de l'alphabet sont égaux et énoncer son théorème de correction.

2.2 Le type prédéfini "option A"

Pour un type A, le type `option A` est

— Soit un élément de A,

— Soit rien.

```
Inductive option (A : Type) : Type :=
| Some : A -> option A
| None : option A
```

Exercice 2 ▸ **Égalité de deux** `option nat`
Définir une fonction `comp_option_nat` qui teste si deux `option nat` sont égaux.

— Par convention, comparer *rien* et *rien* renverra `vrai`.

— Comparer *rien* et *qqchose* renverra forcément `faux`.

— Pour le dernier cas, comparer deux *qqchose* renverra la comparaison effective de ces deux *qqchose*.

Vérifier les tests unitaires et énoncer le théorème de correction associé.

2.3 Le type prédéfini "prod A B"

Le type *produit de A et B* est défini par `prod A B` dans la bibliothèque COQ :

```
Inductive prod (A B : Type) : Type :=
  pair : A -> B -> A * B
```

En Coq, on écrit `A * B` au lieu de `prod A B` (c'est juste une notation).

Ce type n'a qu'un seul constructeur `pair` qui prend deux arguments : `(x:A)` et `(y:B)`. `prod A` est donc *le plus petit ensemble qui contient tous les éléments de la forme* `pair x y` *(avec x dans A et y dans B) et rien d'autre*, donc, intuitivement, `rod A B` est le produit cartésien de A et B, qui contient toutes les paires (x,y) (et rien d'autres).

EXERCICE 3 ▶ **Projection sur les couples d'éléments**
Définir les fonctions `fsta` et `snda` de projection sur les couples d'éléments de type `Alphabet` avec *match* : `p:A*B` correspond au motif (x,y) où x est de type A et y de type B.

EXERCICE 4 ▶ **Comparaison de paires d'entiers**
Définir la fonction `comp_pair_nat` qui compare deux paires d'entiers. L'égalité sur les `nat` est `Nat.eqb` et le connecteur *et* sur les `bool` est `andb`.

EXERCICE 5 ▶ **Swap**
Définir une fonction `swap` qui à la paire d'entiers (a,b) fait correspondre (b,a).

2.4 Recherche dans les listes

Le type des listes prédéfini dans la bibliothèque COQ :

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

Avec les notations :

- `[]` : la liste vide,
- `n::l` : le constructeur d'ajout de n en tête de l,
- `++` : la fonction de concaténation en position infixe.

EXERCICE 6 ▶ **Concaténation**
Définir la fonction `concatene` qui prend en paramètres deux listes d'entiers (donc de type `list nat`) et renvoie la concaténation de ces deux listes.

EXERCICE 7 ▶ **Appartenance**
Définir la fonction `appartient` qui prend en paramètres un entier n et une liste d'entiers (donc de type `list nat`) et renvoie `true` si et seulement si n est dans la liste.

On peut représenter un dictionnaire comme une liste de paire (clef, valeur).

La principale fonctionnalité que l'on attend d'un dictionnaire est de pouvoir retrouver la valeur associée à une clef. Si plusieurs valeurs sont associées, alors on retourne la première qu'on trouve.

On comprend bien que rien ne garantit qu'on trouve toujours une valeur, donc le type de retour de cette fonction est de type `option valeur`.

EXERCICE 8 ▶ **Recherche dans une liste de paires**
Définir la fonction `trouve` qui prend en paramètres

- Une listes de paires (clef,valeur),
- Une clef k,

et renvoie la première valeur associée à k quand elle existe et `None` sinon. Les clés seront des `Alphabet`, les valeurs des `nat`.

2.5 Exercices complémentaires

EXERCICE 9 ▶ **à faire chez vous**
Montrer que `(comp_alphabet x y) = true` *si et seulement si* `(x = y)`.

EXERCICE 10 ▶ **à faire chez vous**
Enoncer et prouver la propriété que comparer un symbole de l'alphabet avec lui-même renvoie vrai.

EXERCICE 11 ▶ **à faire chez vous**
Prouver le lemme suivant :

```
Lemma alphabet_a_juste_deux_elements : forall x:Alphabet, x = a \/ x = b.
```

EXERCICE 12 ▶ **à faire chez vous**
Enoncer et prouver la propriété que la fonction `comp_option_nat` est correcte et complète.

EXERCICE 13 ▶ **à faire chez vous**
Prouver le lemme suivant :

```
Lemma projection_product (A B : Type) : forall p:A*B, p = (fst p, snd p).
```

EXERCICE 14 ▶ **à faire chez vous**
Prouver que `swap` est involutive.
Rappel. Une fonction f est une involution si et seulement si quel que soit x, f(f(x)) = x.

EXERCICE 15 ▶ **à faire chez vous**
Enoncer et prouver la propriété que la fonction `comp_pair_nat` est correcte.

EXERCICE 16 ▶ **à faire chez vous**
Enoncer et prouver la propriété que l'appartenance d'un élément à une liste vide est fausse.

EXERCICE 17 ▶ **à faire chez vous**
Enoncer et prouver la propriété que l'appartenance d'un élément à une liste singleton est vraie si et seulement si l'élément recherché et celui du singleton sont égaux.

EXERCICE 18 ▶ **à faire chez vous**
Énoncer et prouver le lemme de correction de `appartient` nommé `appartient_correct` qui dit en langue naturelle : `(appartient x ls)` est `vrai` *si et seulement si* il existe une décomposition de ls de la forme `ls = l1 ++ x :: l2`.

EXERCICE 19 ▶ **à faire chez vous**
Enoncer et prouver la propriété `trouve_tete` qui, pour toute liste l, toute clé k et toute valeur v,
`trouve ((k,v)::l) k = Some v`.