TP 3
Programmation fonctionnelle et automates en COQ- GALLINA (partie 2)

Fichier fourni : `LF-TP3.v`

Objectifs : Définir des automates et les faire s'exécuter dans la partie *programme* de Coq.

Pour cela, on va utiliser ce qu'on a défini lors du TP2, pour définir les automates :

— Le codage du quintuplet usuel $< K, \Sigma, \delta : K \times \Sigma \rightarrow K, s, F >$ en Coq,

— La représentation finie de la fonction $\delta : K \times \Sigma \rightarrow K$.

Pour aller plus loin, on introduit le polymorphisme en fin de sujet.

On commence par rappeler ce qu'on avait défini dans le TP2.

Notre alphabet d'exemple :

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

La fonction `comp_alphabet` de comparaison de deux Alphabet :

```
Definition comp_alphabet (x y : Alphabet) : bool :=
  match x with
  | a => match y with
         | a => true
         | b => false
         end
  | b => match y with
         | a => false
         | b => true
         end
end.
```

La fonction `appartient` qui teste si un entier appartient à une liste d'entiers :

```
Fixpoint appartient (x : nat) (l : list nat) : bool :=
  match l with
  | [] => false
  | h::rl => (Nat.eqb x h) || (appartient x rl)
  end.
```

La fonction `trouve` qui prend en paramètres une listes de paires `(clef,valeur)` et une clef `k`, et renvoie la première valeur associée à `k` quand elle existe et `None` sinon :

```
Fixpoint trouve (assoc : list (Alphabet * nat)) (key : Alphabet) : option nat :=
  match assoc with
    | [] => None
    | h::rassoc => if (comp_alphabet key (fst h)) then (Some (snd h))
                   else trouve rassoc key
  end.
```

3.1 La représentation des Automates en Coq

Formellement, un Automate est un quintuplet $< K, \Sigma, \delta : K \times \Sigma \to K, s, F >$ avec

- K : l'ensemble des états,
- Σ : l'alphabet,
- δ : la fonction de transition,
- s : l'état initial,
- F : l'ensemble des état finaux.

Ici, on va représenter les ensembles par des listes et la fonction de transition par une fonction (!). On va s'appuyer sur le type `Alphabet` défini dans le TP2. De même, on va prendre les entiers `nat` pour identifier les états.

L'automate `M` défini par `automate K Sigma delta s F`, correspond au quintuplet $\mathrm{M} = < K, \Sigma, \delta, s, F >$ du cours. On justifie la représentation et le choix des types :

- `(K : list nat)` : *liste* de TOUS les états. Une liste c'est différent d'un ensemble, plus facilement programmable.
- `(Sigma : Alphabet)` : liste des symboles utilisés.
- `(delta : nat -> Alphabet -> option nat)`. C'est *presque* le type usuel `K * Sigma -> K` à la curryfication près ET avec une *option* sur le résultat. `option` permet d'exprimer que la fonction de transition delta est *partielle* (et non totale) : on va en fait manipuler en Coq des automates aux transitions *partielles*.
- `(s : nat)` : état initial.
- `(F : list nat)` : une liste de `nat` des états finals. Là encore ensemble $\neq$ liste.

EXERCICE 1 ► **Type Automate**
Définir le type `Automate` représentant ce quintuplet. Ce type aura un seul constructeur que l'on nommera `automate`.

EXERCICE 2 ► **Accesseurs d'automates**
Définir les 5 fonctions suivantes :

- `etats` : prend en paramètre un automate et renvoie la liste des états,
- `symboles` : prend en paramètre un automate et renvoie la liste des symboles de l'alphabet,
- `initial` : prend en paramètre un automate et renvoie l'état initial,
- `acceptant` : prend en paramètre un automate et un état q et renvoie true ssi q est un état final,
- `transition` : prend en paramètre un automate, un état q et un symbole c, et renvoie l'état (optionnellement) accessible depuis q en lisant c.

EXERCICE 3 ► **Exemple 1 : nombre de b impair**
Soit l'automate `M_nb_b_impair` à deux états qui accepte les mots contenant un nombre impair de `b`. La fonction delta est donnée ci-dessous :

```
Definition delta_nb_b_impair (q : nat) (c : Alphabet) : option nat :=
match (q,c) with
 | (1,a) => Some 1
 | (1,b) => Some 2
 | (2,a) => Some 2
 | (2,b) => Some 1
 | (_,_) => None
end.
```

- DESSINER L'AUTOMATE `M_nb_b_impair`,
- Définir `M_nb_b_impair`,
- Donner des tests unitaires.

EXERCICE 4 ▸ **Fonction execute**
Définir La fonction `execute` qui prend en paramètre un automate, un état q et un mot w (une `list Alphabet`), et qui va calculer l'état d'arrivée, en partant de l'état q et en lisant le mot w.

EXERCICE 5 ▸ **Fonction reconnait**
Définir la fonction `reconnait` qui prend en paramètre un automate et un mot w, et qui renvoie vrai si w est accepté par l'automate, faux sinon.

EXERCICE 6 ▸ **Exemple 2 : commence et finit par a**
Soit l'automate `M_commence_et_finit_par_a` à trois états qui accepte les mots commençant et finissant par `a`,

— DESSINER L'AUTOMATE `M_commence_et_finit_par_a`,
— Définir `M_commence_et_finit_par_a`,
— Donner des tests unitaires,
— Tester l'automate avec les fonctions `execute` et `reconnait`.

3.2 La représentation des fonctions de transition en Coq ou recherche dans les listes de paires

On souhaite donner une description de la fonction de transition par SON GRAPHE plutôt que donner son code.

Rappel, le graphe d'une fonction f : A -> B est la relation définie par (x,f(x)) | x dans A.

Par exemple la liste [((1,a),1) ; ((1,b),2) ; ((2,a),2) ; ((2,b),1)] indique que

— ((1,a),1) : état courant 1, symbole courant a –> nouvel état 1
— ((1,b),2) : état courant 1, symbole courant b –> nouvel état 2
— ((2,a),2) : état courant 2, symbole courant a –> nouvel état 2
— ((2,b),1) : état courant 2, symbole courant b –> nouvel état 1

Cette liste *recopie* la fonction `delta_nb_b_impair` donnée ci-dessus.

Comme le domaine de la fonction de transition est fini, on peut faire l'inverse, c'est-à-dire construire une fonction à partir d'un graphe FINI.

On va représenter le graphe de f par un *dictionnaire*, c'est-à-dire une liste de paires (clé, valeur).

La principale fonctionnalité que l'on attend d'un dictionnaire est de pouvoir retrouver la valeur associée à une clé. En le faisant, on reconstruit (à un *option* près) f.

EXERCICE 7 ▸
Définir la fonction `trouve_paire` avec pour type `list ((nat * Alphabet) * nat) -> (nat * Alphabet) -> option nat` qui prend en paramètres une liste et une clé et retourne la première valeur correspondant à la clé si elle existe, None sinon.

La liste est une liste de `((nat * Alphabet) * Alphabet)` et donc la clé est un `(nat * Alphabet)`.

EXERCICE 8 ▸
En utilisant `trouve_paire`, définir une fonction `graphe_vers_fonction` qui transforme une liste `list ((nat * Alphabet) * nat)` en une fonction `nat -> Alphabet -> option nat`.

EXERCICE 9 ▸ **Exemple 1 : nombre de b impair**
Définir l'automate `M_nb_b_impair'` à deux états qui accepte les mots contenant un nombre impair de 'b', et donner des tests unitaires. Le graphe de transition est donnée ci-dessous.

```
Definition graphe_nb_b_impair := [((1,a), 1) ; ((1,b),2) ; ((2,a),2) ; ((2,b),1)].
```

EXERCICE 10 ▸ **Exemple 2 : commence et finit par a**
Définir l'automate à trois états qui accepte les mots commençant et finissant par 'a', et donner des tests unitaires. Définir pour cela le graphe puis l'automate qui l'utilise.

EXERCICE 11 ►
Rappel : dans `M_nb_b_impair'` et `M_nb_b_impair` on ne s'intéresse qu'aux états, PAS à TOUS les entiers, donc on met une garde sur q.

Montrer que `delta_nb_b_impair` et `delta_nb_b_impair_graphe` sont équivalents sur les états valides.

Cette preuve utilise la logique du premier ordre, nous reviendrons dessus plus tard.

3.3 Pour aller plus loin : le polymorphisme

Quand on lit et a fortiori quand on écrit la fonction `appartient`, on remarque son caractère générique sur les listes..

Elle est écrite pour le type `list nat` mais si on remplace `Nat.eqb` par une fonction `comp_A : A -> A -> bool`, `appartient` fonctionnerait pour un type donné `A`.

EXERCICE 12 ► **à faire chez vous**
Définir la fonction `appartient_poly` qui prend en paramètres

— Un type `A`,
— Une fonction `comp_A` de décision de l'égalité sur `A`,
— Un élement `x` de type `A`,
— Une liste `l` d'éléments de `A`,

et renvoie true si et seulement si l'élément x est dans la liste l.

EXERCICE 13 ► **à faire chez vous**
Montrer que `appartient` est juste l'instance particulière de `appartient_poly nat (Nat.eqb) nat (Nat.eqb)`.

Pour bien représenter *l'appartenance* à la liste, il faut quand même s'assurer que `comp_A` respecte la spécification *décider de l'égalité dans A*. Les exemples suivants montrent des choix arbitraires de `comp_A` :

```
Example appartient_poly_ex3 : appartient_poly nat (fun x y => false) 0 [1;3;0;5] = false.
Example appartient_poly_ex4 : appartient_poly nat (fun x y => true) 4 [1;3;0;5] = true.
```

Si on veut prouver le lemme équivalent pour `appartient_poly`, on a besoin d'une propriété de type `forall x y:A, comp_A x y = true <-> x = y` similaire à `PeanoNat.Nat.eqb_eq`, `comp_alphabet_eq`, `comp_option_nat_correct`, etc.

EXERCICE 14 ► **à faire chez vous**
Montrer que si x = y alors x appartient à une liste constituée que de [y].

On peut même aller plus loin et montrer que `(comp x y = true <-> x = y)` est non seulement SUFFISANTE mais aussi NECESSAIRE si on veut `appartient_poly A comp x [y] = true <-> x = y`.

EXERCICE 15 ► **à faire chez vous**
Montrer que si x appartient à une liste constituée que de [y] alors x = y.

EXERCICE 16 ► **à faire chez vous**
Définir la fonction `trouve_poly`, version polymorphe de `trouve` et `trouve_paire`.

EXERCICE 17 ► **à faire chez vous**
Montrer que `trouve` et `trouve_paire` sont bien des instances de `trouve_poly`.