INF3038L

LF : Langages formels

TP

2025 – 2026

Table des matières

LF TP 1

Programmation fonctionnelle en COQ- GALLINA

Fichier fourni : `lf_tp1.v`

Objectifs :

— Se familiariser avec l'environnement COQIDE et le langage de programmation GALLINA,

— Se familiariser avec 4 mots-clés : `Definition`, `Definition` avec `match ... with`, `Inductive`, `Fixpoint`.

EXERCICE 1 ▶ **Mise en route**
Téléchargez `lf_tp1.v` et ouvrez le avec COQIDE. Vous pouvez également l'ouvrir avec VISUAL STUDIO CODE si vous avez installé COQ sur votre système et l'extension VSCOQ de VSCODE. Vous écrirez votre code et le compilerez directement dans ce fichier, qui reprend le canevas de ce document.

Définir un objet (entier, fonction ...) : Mot-clé `Definition`
`Definition` <nom de l'objet> : <type de l'objet> := <valeur de l'objet> .

```
Definition a : nat := 3.
Definition b : nat := 6.
```

Effectuer un calcul dans l'interpréteur : directive `Compute`

```
Compute (a+b).
```

Afficher le type dans l'interpréteur : directive `Check`

```
Check (a+b).
```

Afficher la valeur dans l'interpréteur : directive `Print`

```
Print a.
```

1.1 Types énumérés et inductifs

Définition d'un ensemble inductif : Mots-clés `Inductive` et | (pipe) par cas
On donne des règles. Comme on définit un *type* de données, son propre type est `Type`.

```
Inductive jour : Type :=
  | lundi : jour
  | mardi : jour
  | mercredi : jour
  | jeudi : jour
  | vendredi: jour
  | samedi : jour
  | dimanche : jour.
```

Définition d'une fonction : Mots-clés `Definition`, `match`, `with` et `end`
Réalisée suivant *la forme* du paramètre, c'est du *filtrage de motif* ou *pattern matching*. C'est le mécanisme le plus confortable pour manipuler des structures inductives.

```
Definition jour_suivant (j : jour) : jour :=
  match j with
  | lundi => mardi
  | mardi => mercredi
  | mercredi => jeudi
  | jeudi => vendredi
  | vendredi => samedi
  | samedi => dimanche
  | dimanche => lundi
  end.
```

EXERCICE 2 ▶
Définir la fonction qui retourne le surlendemain d'un jour donné. C'est une fonction qui **appliquée à** un jour, **retourne** un jour.

Les booléens

```
Inductive booleens : Type :=
| Vrai : booleens
| Faux : booleens.

Definition non (a : booleens) : booleens :=
  match a with
  | Vrai => Faux
  | Faux => Vrai
  end.
```

EXERCICE 3 ▶
Définir la fonction *et* sur les booléens.

EXERCICE 4 ▶
Définir la fonction *ou* sur les booléens.

EXERCICE 5 ▶
Définir une fonction `bcompose : f -> g -> h` telle que `h` est la composition des deux fonctions booléennes `f : booleens -> booleens` et `g : booleens -> booleens`.
Tester `bcompose` en définissant une fonction `nonnon : booleens -> booleens` qui définit `non o non`.

Le langage de Coq a bien sûr des booléens (dans le type prédéfini bool), ils sont en fait définis de la même façon que nos booléens. Pour l'instant nous allons continuer de travailler avec les nôtres.

Les entiers

On définit maintenant de façon inductive le type des entiers naturels. Un entier naturel est :

— Soit un élément particulier noté Z (pour zéro, c'est un cas de base ici),

— Soit le successeur d'un entier naturel.

On a bien deux constructeurs pour les entiers : ils sont soit de la *forme* "Z" soit de la *forme* "Succ d'un entier".

```
Inductive entiers : Type :=
| Z : entiers
| Succ : entiers -> entiers.

Definition un  := Succ Z.
Definition deux  := Succ un.
Definition trois  := Succ deux.
```

EXERCICE 6 ▶
Définir la fonction prédécesseur `pred`. C'est une fonction qui **appliquée à** un entier, **retourne** un entier. Par convention, le prédécesseur de `Z` est `Z`.

Définition d'une fonction récursive : Mot-clé `Fixpoint`

On veut écrire une fonction récursive pour ajouter deux entiers. Comme la fonction est récursive, on utilise le mot-clé `Fixpoint` (et non plus `Definition`). Elle se calcule selon la forme du premier paramètre.

```
Fixpoint plus (a : entiers) (b : entiers) : entiers :=
  match a with
  | Z => b
  | Succ n => Succ (plus n b)
  end.
```

EXERCICE 7 ▶
Définir la fonction `mult` qui calcule le produit de deux entiers. Elle se calcule selon la forme du premier paramètre.

EXERCICE 8 ▶
Définir une fonction `est_pair`, telle que `est_pair` appliquée un entier `a` retourne `Vrai` si `a` est pair, `Faux` sinon.

EXERCICE 9 ▶
Définir la fonction `factorielle` sur les `entiers`.

EXERCICE 10 ▶
Définir la fonction `moins`, soustraction non négative sur les `entiers`.

EXERCICE 11 ▶
Définir une fonction `inf`, tel que `inf a b` vaut / retourne `Vrai` si `a` est inférieur ou égal à `b`, `Faux` sinon.

EXERCICE 12 ▶
Définir une fonction `egal`, tel que `egal a b` donne `Vrai` si les entiers `a` et `b` sont égaux, `Faux` sinon.

Types prédéfinis

Précédemment, on a défini nos booléens et nos entiers naturels, mais ils sont en fait déjà définis dans la bibliothèque que COQ charge initialement au démarrage :

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

avec les fonctions `negb` (complémentaire), `andb` (et, min), `orb` (ou, max).

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

avec les fonctions usuelles `+` , `-` , `*` , etc. et les comparaisons : `Nat.eqb` pour le test d'égalité, `Nat.ltb` pour le test plus petit, `Nat.leb` pour le test plus petit ou égal.

Pour la suite, lorsque cela n'est pas précisé, nous utiliserons les booléens et entiers prédéfinis.

1.2 Arbres binaires

EXERCICE 13 ▸
Donner une définition par induction de l'ensemble `nBin` des arbres binaires contenant des `nat`.
Deux constructeurs :

— `nEmpty` : arbre vide,

— `nNode` : création d'un noeud avec un fils gauche, un `nat` et un fils droit.

Exemple d'arbre à 5 éléments :

```
Definition a1 := nNode
                      (nNode nEmpty 2 nEmpty)
                      1
                      (nNode
                            (nNode nEmpty 4 nEmpty)
                            3
                            (nNode nEmpty 5 nEmpty)
                      ).

Check a1.
Print a1.
```

EXERCICE 14 ▸
Définir la fonction `nelements` qui renvoie la liste des éléments contenus dans un arbre binaire de `nat`. Le faire naïvement avec un `concat` pour commencer.

EXERCICE 15 ▸
Définir la fonction `nnelts` qui renvoie le nombre de noeuds internes (portant une étiquette de type `nat`) dans un `nBin`.

EXERCICE 16 ▸
Définir la fonction `nfeuilles` qui renvoie le nombre de feuilles d'un `nBin`.

EXERCICE 17 ▸
Définir la fonction `nsum` qui renvoie la somme des valeurs portées par les noeuds internes d'un `nBin`.

LF 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`.

LF 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 $< K, \Sigma, \delta : K \times \Sigma \to K, s, F >$ en Coq,

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

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

3.1 Rappel du TP2

Notre alphabet d'exemple :

```
Inductive Alphabet : Type :=
| a : Alphabet
| b : Alphabet
| c : Alphabet. (* On ajoute c au cas où, même si on ne l'utilisera pas dans les exemples suivants
```

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
         | c => false
         end
  | b => match y with
         | a => false
         | b => true
         | c => false
  end
  | c => match y with
         | a => false
         | b => false
         | c => true
         end
  end.
```

La preuve de correction de la fonction de comparaison de deux alphabets :

```
Lemma comp_alphabet_correct : forall x y, comp_alphabet x y = true -> x = y.
Proof.
  intro x.
  intro y.
  intro Hcomptrue.
  destruct x.
  - destruct y.
    + reflexivity.
    + cbv in Hcomptrue. discriminate.
    + cbv in Hcomptrue. discriminate.
  - destruct y.
    + cbv in Hcomptrue. discriminate.
    + reflexivity.
    + cbv in Hcomptrue. discriminate.
 - destruct y.
    + cbv in Hcomptrue. discriminate.
    + cbv in Hcomptrue. discriminate.
    + reflexivity.
Qed.
```

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

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

La fonction `appartient_alphabet` qui teste si un Alphabet appartient à une liste d'Alphabet :

```
Fixpoint appartient_alphabet (s : Alphabet) (l : list Alphabet) : bool :=
  match l with
  | [] => false
  | h::rl => (comp_alphabet s h) || (appartient_alphabet s 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.
```

3.2 La représentation des Automates en Coq

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

— K : ensemble des états,

— Σ : alphabet (ensemble des symboles utilisés par l'automate),

— δ : fonction de transition,

— s : état initial,

— F : 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 `nat` pour identifier les états.

L'automate `M` défini par `automate K Sigma delta s F`, correspond au quintuplet 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 diffère d'un ensemble, plus facilement programmable.

— (`Sigma : list 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 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 ≠ liste.

EXERCICE 1 ▶ **Type Automate**
Définir le type `Automate` représentant ce quintuplet. Ce type aura un seul constructeur que nommé `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) (s : Alphabet) : option nat :=
match q with
 | 1 => match s with
        | a => Some 1
        | b => Some 2
        | _ => None
        end
 | 2 => match s with
        | a => Some 2
        | b => Some 1
        | _ => None
        end
 | _ => 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`.

3.3 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 ▶ **Preuve d'équivalence des fonctions** `delta`
Montrer que `delta_nb_b_impair` et `delta_nb_b_impair_graphe` sont équivalents sur les états valides et les symboles utilisés.

3.4 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`, syntaxe : `(A : Type)`
- Une fonction `comp_A` de décision de l'égalité sur `A`, syntaxe : `(comp_A : A -> A -> bool)`
- 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`.

LF TP 4

Grammaires et automates en COQ

Fichier fourni : `lf_tp4.v`

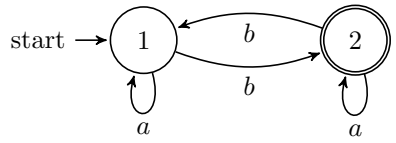
Objectifs : À partir de la définition des automates vue au TP précédent,

— Ajouter la définition d'une grammaire équivalente,

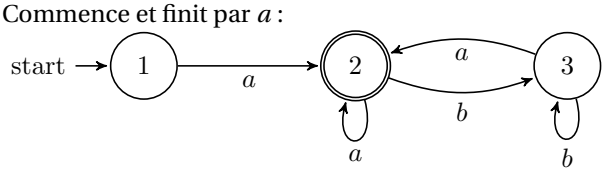
— Prouver que l'automate et la grammaire sont effectivement équivalents.

On va utiliser les deux automates définis au TP précédent :

Nombre de b impair :



Commence et finit par a :



4.1 Rappel du TP précédent

Le type `Automate` :

```
Inductive Automate : Type :=
  automate : list nat -> list Alphabet -> (nat -> Alphabet -> option nat) -> nat -> list nat
               -> Automate.
```

La fonction `etats` qui prend en paramètre un automate et renvoie la liste des états :

```
Definition etats (M : Automate) :  list nat :=
  match M with
    automate ql _ _ _ _ => ql
  end.
```

La fonction `symboles` qui prend en paramètre un automate et renvoie la liste des symboles de l'alphabet :

```
Definition symboles (M : Automate) :  list Alphabet :=
  match M with
    automate _ sigma _ _ _ => sigma
  end.
```

La fonction `nitial` qui prend en paramètre un automate et renvoie l'état initial :

```
Definition initial  (M : Automate) :  nat :=
  match M with
    automate _ _ _ q0 _ => q0
  end.
```

La fonction `acceptant` qui prend en paramètre `M` et q et renvoie `true` ssi q est un état acceptant de `M` :

```
Definition acceptant  (M : Automate) (q : nat) : bool  :=
  match M with
    automate _ _ _ _ lF => (appartient q lF)
  end.
```

La fonction `transition` qui prend en paramètre un automate, un état q et un symbole c, et renvoie l'état (optionnellement) accessible depuis q en lisant c :

```
Definition transition  (M : Automate) (q : nat) (c : Alphabet) : option nat :=
  match M with
    automate _ _ f _ _ => f q c
  end.
```

La fonction `execute` qui va calculer l'état d'arrivée en lisant un mot, c'est-à-dire une `list Alphabet` :

```
Fixpoint execute (M : Automate)  (q : nat) (w : list Alphabet) : option nat :=
  match w with
  | [] => Some q
  | h::rw => match transition M q h with
             | None => None
             | Some e => execute M e rw end
  end.
```

La fonction `reconnait` qui va accepter ou refuser un mot :

```
Definition reconnait (M : Automate) (w : list Alphabet) : bool :=
  match (execute M (initial M) w) with
  | None => false
  | Some e => acceptant M e
  end.
```

L'automate `M_nb_b_impair` à deux états qui accepte les mots contenant un nombre impair de b :

```
match q with
 | 1 => match s with
           | a => Some 1
           | b => Some 2
           | _ => None
           end
 | 2 => match s with
           | a => Some 2
           | b => Some 1
           | _ => None
           end
 | _ => None
end.
Definition M_nb_b_impair := automate [1;2] [a;b] (delta_nb_b_impair) 1 [2].
```

L'automate `M_commence_et_finit_par_a` à trois états qui accepte les mots commençant et finissant par a :

```
Definition delta_commence_et_finit_par_a (q : nat) (s : Alphabet) : option nat :=
match q with
| 1 => match s with
          | a => Some 2
          | _ => None
          end
| 2 => match s with
          | a => Some 2
          | b => Some 3
          | _ => None
          end
| 3 => match s with
          | a => Some 2
```

```
          | b => Some 3
          | _ => None
          end
| _ => None
end.
Definition M_commence_et_finit_par_a :=
    automate [1;2;3] [a;b] (delta_commence_et_finit_par_a) 1 [2].
```

4.2 Grammaire implémentée par un automate

L'automate `M_nb_b_impair` implémente la grammaire `G_nb_b_impair` suivante :

$S_1 \to aS_1|bS_2$
$S_2 \to aS_2|bS_1|\epsilon$

`G_nb_b_impair w i` : le PRÉDICAT "mot généré par `G_nb_b_impair` à partir du non-terminal S_i".

```
Inductive G_nb_b_impair : (list Alphabet) -> nat -> Prop :=
| G_nb_b_impair_0 : G_nb_b_impair [] 2
| G_nb_b_impair_1a : forall w, G_nb_b_impair w 1 -> G_nb_b_impair (a::w) 1
| G_nb_b_impair_1b : forall w, G_nb_b_impair w 2 -> G_nb_b_impair (b::w) 1
| G_nb_b_impair_2a : forall w, G_nb_b_impair w 2 -> G_nb_b_impair (a::w) 2
| G_nb_b_impair_2b : forall w, G_nb_b_impair w 1 -> G_nb_b_impair (b::w) 2.
```

EXERCICE 1 ▸
Comprendre et expliquer en langue naturelle comment est construit et comment fonctionne ce prédicat.

`G_nb_b_impair` génère le mot `abaabab` à partir du non-terminal S_1.

```
Example ex_G_nb_b_impair_1 : G_nb_b_impair [a;b;a;a;b;a;b] 1.
Proof.
  apply G_nb_b_impair_1a. (* état courant 1, symbole courant a -> état 1, reste baabab *)
  apply G_nb_b_impair_1b. (* état courant 1, symbole courant b -> état 2, reste aabab *)
  apply G_nb_b_impair_2a. (* état courant 2, symbole courant a -> état 2, reste abab *)
  apply G_nb_b_impair_2a. (* état courant 2, symbole courant a -> état 2, reste bab *)
  apply G_nb_b_impair_2b. (* état courant 2, symbole courant b -> état 1, reste ab *)
  apply G_nb_b_impair_1a. (* état courant 1, symbole courant a -> état 1, reste b *)
  apply G_nb_b_impair_1b. (* état courant 1, symbole courant b -> état 2, reste epsilon *)
  apply G_nb_b_impair_0.
Qed.
```

`G_nb_b_impair` génère le mot `baab` à partir du non-terminal S_2 :

```
Example ex_G_nb_b_impair_2 : G_nb_b_impair [b;a;a;b] 2.
Proof.
  apply G_nb_b_impair_2b.
  apply G_nb_b_impair_1a.
  apply G_nb_b_impair_1a.
  apply G_nb_b_impair_1b.
  apply G_nb_b_impair_0.
Qed.
```

Evidemment, `G_nb_b_impair` ne peut pas générer, par exemple, le mot `bab` à partir du non-terminal S_1.

EXERCICE 2 ▸
Définir la grammaire `G_commence_et_finit_par_a` implémentée par l'automate `M_commence_et_finit_par_a`, et donner des exemples de mots générés par cette grammaire.

4.3 Equivalence grammaire et automate

On veut prouver :
Soit un automate `M` qui implémente une grammaire `G`. `G` génère un mot w à partir de S_1 ssi `M` accepte w.
En particulier :

— `G_nb_b_impair` génère un mot w à partir de S_1 ssi `M_nb_b_impair` accepte w,

— `G_commence_et_finit_par_a` génère un mot w à partir de S_1 ssi `M_commence_et_finit_par_a` accepte w.

4.3.1 Sens Automate ⇒ Grammaire

On sait trouver une exécution par dérivation :
Si `G` permet de générer un mot w à partir du non-terminal S_q, alors `M` accepte w à partir de l'état q.

EXERCICE 3 ▸
Montrer que si `G_nb_b_impair` génère un mot w à partir du non terminal S_q, alors `M_nb_b_impair` accepte ce même mot w à partir de l'état q.

```
Lemma G_nb_b_impair_mime_M_nb_b_impair :
  forall w q, G_nb_b_impair w q
  -> exists e, execute M_nb_b_impair q w = Some e /\ acceptant M_nb_b_impair e = true.
```

EXERCICE 4 ▸ **à faire chez vous**
Même exercice avec la grammaire `G_commence_et_finit_par_a`.

Tout mot w généré par `G` à partir de la source est reconnu par `M`.
Si `G` génère un mot w à partir du non-terminal S_1, alors `M` accepte w.

EXERCICE 5 ▸
Montrer que si `G_nb_b_impair` génère un mot w, alors `M_nb_b_impair` accepte w.

```
Lemma G_nb_b_impair_reco_M_nb_b_impair :
  forall w, G_nb_b_impair w 1 -> reconnait M_nb_b_impair w = true.
```

4.3.2 Sens Grammaire ⇒ Automate

On sait trouver une dérivation par exécution.
Si q est un état valide de `M` alors si `M` accepte un mot w à partir de l'état q, alors `G` génère ce même mot w à partir de son non-terminal S_q.

EXERCICE 6 ▸
Montrer que si `M_nb_b_impair` accepte un mot w à partir d'un état valide, alors `G_nb_b_impair` génère w.

```
Lemma M_nb_b_impair_mime_G_nb_b_impair :
  forall w q, (appartient q (etats M_nb_b_impair)) = true
  -> (forall e, execute M_nb_b_impair q w = Some e /\ acceptant M_nb_b_impair e = true
        -> G_nb_b_impair w q).
```

Remarque : le théorème précédent pourrait être formulé de la manière suivante : voici le corollaire.

```
Lemma M_nb_b_impair_mime_G_nb_b_impair' :
  forall w q, (appartient q (etats M_nb_b_impair)) = true
  /\ (forall e, execute M_nb_b_impair q w = Some e /\ acceptant M_nb_b_impair e = true)
        -> G_nb_b_impair w q.
Proof.
intros w q H.
destruct H as [H1 H2].
```

```
apply M_nb_b_impair_mime_G_nb_b_impair with (e := 2).
- exact H1.
- apply H2.
Qed.
```

EXERCICE 7 ► **à faire chez vous**
Même exercice avec l'automate `M_commence_et_finit_par_a`.

Tout mot w reconnu par `M` est généré par `G` à partir de la source.
Si `M` accepte le mot w, alors `G` génère ce même mot w à partir du non-terminal S_1.

EXERCICE 8 ►
Montrer que si `M_nb_b_impair` accepte un mot w, alors `G_nb_b_impair` génère w à partir de S_1.

```
Lemma M_nb_b_impair_regu_G_nb_b_impair :
  forall w, reconnait M_nb_b_impair w = true -> G_nb_b_impair w 1.
```