TP 4

Grammaires et automates en Coq

Fichier fourni : `LF-TP4.v`

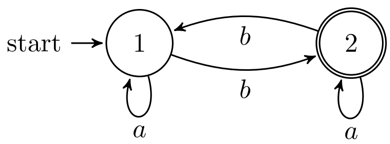
Objectifs : À partir de la définition des automates vue au TP précédent,

— Ajouter la définition d'une grammaire équivalente,

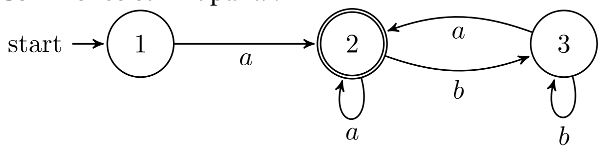
— Prouver que l'automate et la grammaire sont effectivement équivalents.

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 :

```
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.
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) (c : Alphabet) : option nat :=
match (q,c) with
 | (1,a) => Some 2
 | (2,a) => Some 2
 | (2,b) => Some 3
 | (3,a) => Some 2
 | (3,b) => Some 3
 | (_,_) => 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.
```