Exercices de Bases formelles du TAL

Pierre-Léo Bégay

January 29, 2020

Grammaires

Nettoyage de grammaire

Soit la grammaire suivante :

- $S \to AA$
- $A \to aA \mid Aa \mid bB$
- $B \to AaA \mid \epsilon$
- $C \to SA \mid BD$
- $D \to DaD \mid \epsilon$
- $E \to AS \mid CD \mid aE$

Question 1 De quel type est cette grammaire ?

Correction Il ne peut s'agir d'une grammaire de type 3 à cause (notamment) des nombreuses règles avec deux non-terminaux à droite de la flèche. Il s'agit par contre bien d'une grammaire de type 2, puisqu'à gauche de toute flèche, on a systématiquement un non-terminal et rien d'autre.

Question 2 Nettoyez la grammaire.

Correction On rappelle qu'une grammaire propre est une grammaire

- sans ϵ-production
- sans cycle ($A \to^+ A$)
- sans symbole inutile, cad
  - sans symbole sans production
  - sans symbole inaccessible

L'élimination de symboles inutiles est plus simple / rapide au début, et peut permettre de simplifier les calculs des étapes suivantes. On commence donc par ça :

Elimination des symboles sans production On calcule les symboles productifs en utilisant l'algorithme de point fixe vu en cours. On voit que B et D produisent des "vrais mots" (bien que vides), on a donc $N_1 = \{B, D\}$.

On cherche maintenant les symboles qui produisent des mots composés de terminaux et de B ou D. On a donc $N_2 = N1 \cup \{A, C\} = \{A, B, C, D\}$.

On cherche maintenant les symboles qui produisent des mots composés de terminaux et de A, B, C ou D. On a donc $N_3 = N2 \cup \{S, E\} = \{S, A, B, C, D, E\}$. Puisque tous les non-terminaux sont productifs, on ne supprime rien.

Elimination des symboles inaccessibles On calcule les symboles accessibles en utilisant l'algorithme de point fixe vu en cours. De base, l'axiome S est bien évidemment accessible. S produit des A, on a donc $N_2 = \{S, A\}$.

A produit un B. On a donc $N_3 = \{S, A, B\}$. Par contre, le seul non-terminal produit par B est A, qu'on avait déjà noté comme accessible. La nouvelle itération ne change rien, on a donc $N_4 = N_3$ et le point fixe est atteint. Le complémentaire de N_4, c'est-à-dire $\{C, D, E\}$ est donc inaccessible. La nouvelle grammaire est donc

- $S \to AA$
- $A \to aA \mid Aa \mid bB$
- $B \to AaA \mid \epsilon$

Remarque Pas d'ordre particulier entre l'élimination de symboles sans production et inaccessibles. Ici, l'ordre a été volontairement mal choisi pour avoir plus de calculs avec les symboles sans production.

Remarque bis Pour pouvoir mieux s'entraîner sur les autres algorithmes, on va supposer qu'on n'a pas cherché les symboles inaccessibles et se baser sur la grammaire initial.

Remarque ter Pour éliminer les cycles, on suppose l'absence d'ϵ-productions. On commence donc par ça.

Elimination d'ϵ-productions Si on se contente d'éliminer les ϵ-productions, la grammaire obtenue risque d'avoir perdu de son pouvoir expressif (cad. des mots qu'elle engendre). Par exemple, sans les ϵ-productions, on ne pourrait plus engendrer le mot bb (vérifiez ça). La stratégie qu'on a vue en cours était de repérer tous les non-terminaux qui peuvent, éventuellement en plusieurs étapes, se réécrire en ϵ, et "potentiellement effacer" toutes leurs occurrences.

Pour la première étape, on procède également par point fixe. On remarque d'abord que B et D se réécrivent immédiatement en ϵ. Ensuite, on note que C se réécrit en BD. Puisque chaque non-terminal de ce mot peut se réécrire en ϵ, c'est également le cas de C. On cherche ensuite un non-terminal qui se réécrirait en un mot composé uniquement de B, C et D. C'est le cas de E, via la règle $E \to CD$, il est donc ajouté à notre ensemble. On ne trouve ensuite pas d'autre non-terminal se réécrivant en un mélange de B, C, D et E, le point fixe est donc atteint.

On va maintenant prendre chacune des règles de la grammaire, et, pour chaque occurrence à droite d'une flèche d'un B, C, D ou E, dupliquer la règle avec ou sans l'occurrence en question :

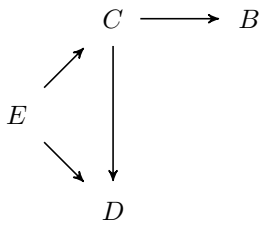
- Puisque B ne se réécrit plus en ϵ, pour éviter de perdre la possibilité $A \to^* b$, on ajoute précisément cette règle.
- Pour la règle $C \to BD$, on veut permettre que B se réécrive en ϵ mais pas D, et inversement. On rajoute donc les deux règles $C \to B \mid D$.
- Pour la règle $D \to DaD$, on veut que chacun des D se réécrive en ϵ ou, au contraire, fasse une "vraie dérivation", et ceci indépendamment l'un de l'autre. On créé donc 3 règles supplémentaires : $D \to aD \mid Da \mid a$.
  - Notez que pour $C \to B \mid D$, on n'a pas rajouté la possibilité où B et D se réécrivent tous les deux en ϵ, car ça nous aurait fait ajouter la règle $C \to \epsilon$, ce qu'on veut justement éviter ! On ne perd quand même pas de pouvoir expressif, car chaque C sera également "potentillement remplacé" par un ϵ, vu qu'il a été "repéré" par la première étape
- Enfin, on ajoute les règles $E \to C \mid D \mid a$

Au final, on obtient la grammaire suivante :

- $S \to AA$
- $A \to aA \mid Aa \mid bB \mid b$
- $B \to AaA$
- $C \to SA \mid BD \mid B \mid D$
- $D \to DaD \mid aD \mid Da \mid a$
- $E \to AS \mid CD \mid aE \mid C \mid D \mid a$

Elimination de cycles Maintenant qu'on n'a plus d'ϵ-productions, une dérivation ne peut pas réduire un terme, cad. si on a $u \to^* v$, alors $|u| \leq |v|$. Si on a un cycle, ça veut dire qu'à chaque étape, on passe d'un non-terminal à un autre, par exemple $A \to B \to C \to A$. En éliminant les transitions unitaires, cad. celles de la forme $A \to B$, on est donc sûr d'éviter les cycles. Il faut cependant bien sûr préserver le langage engendré.

Le but du jeu est ici d'identifier quels non-terminaux peuvent se réécrire en lesquels. On voit immédiatement que C se réécrit en B et D. De plus, E se réécrit en D et C. Via cette dernière transition, E se réécrit également en B. Pour résumer :



Maintenant, pour chaque non-terminaux A et B tels que $A \to^+ B \to u$ où u n'est pas un non-terminal, on veut rajouter la transition $A \to u$. En l'occurrence, la grammaire devient

- $S \to AA$
- $A \to aA \mid Aa \mid bB \mid b$
- $B \to AaA$
- $C \to SA \mid BD \mid AaA \mid DaD \mid aD \mid Da \mid a$
- $D \to DaD \mid aD \mid Da \mid a$
- $E \to AS \mid CD \mid aE \mid a \mid SA \mid BD \mid AaA \mid DaD \mid aD \mid Da \mid a$

Transformations automates / grammaires

Soit la grammaire suivante :

- $S \to AS \mid \epsilon$
- $A \to aA \mid aB$
- $B \to bB \mid b$

Question 1 Quel est le langage engendré ?

Correction On va étudier les langages engendrés par chacun des non-terminaux, en partant des moins dépendants :

- B génère un certain nombre de b puis termine avec un b. Le langage engendré par B est donc b^+
- A génère un certain nombre de a puis termine avec aB. Le langage engendré par A est donc a^+b^+
- S génère un certain nombre de A puis termine avec un ϵ. Le langage engendré (tout court) est donc $A^* = (a^+b^+)^*$

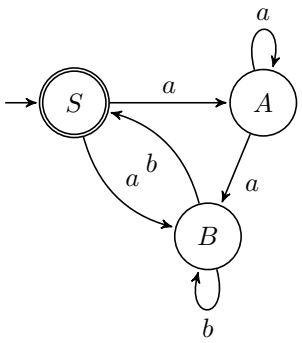
Question 2 La grammaire ci-dessus est de type 2 (cf. la règle $S \to AS$), et pourtant on a pu exprimer le langage qu'elle engendre via une expression rationnelle. Ces dernières sont normalement équivalentes aux grammaires de type 3 et pas plus, comment expliquer ce paradoxe ?

Correction La grammaire utilisée est en fait plus puissante que nécessaire. En effet, bien qu'elle soit de type 2, le langage généré est bien de type 3 et peut être exprimé avec une grammaire du même type :

- $S \to aA \mid aB \mid \epsilon$
- $A \to aA \mid aB$
- $B \to bB \mid bS$

Question 3 Transformez la grammaire de la question précédente en automate

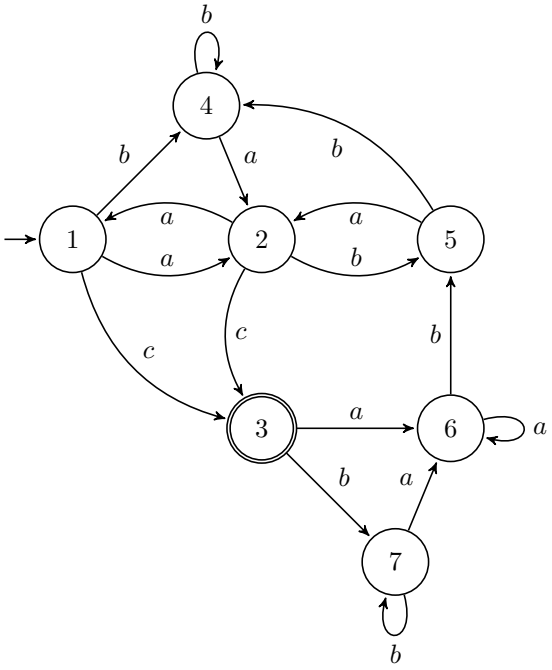
Correction En appliquant la correspondance "état = symbole non-terminal" :



Automates

Regex → automate

Soit l'automate suivant :



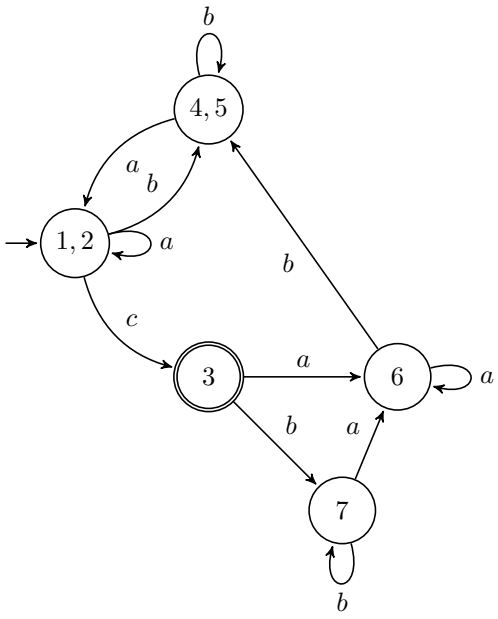
Question 1 Minimisez-le.

On pose $C_T = \{3\}$ et $C_{nT} = \{1, 2, 4, 5, 6, 7\}$. Puisque le seul élément à ne pas être dans C_{nT} est 3, la seule façon de "casser" C_{nT} est de trouver des états qui vont dans 3 et d'autres qui, avec les mêmes lettres, n'y vont pas. On remarque immédiatement que 1 et 2 vont en 3 avec c, et c'est tout. On peut donc casser C_{nT} en $C_\alpha = \{1, 2\}$ et $C_\beta = \{4, 5, 6, 7\}$.

Si on regarde 1 et 2, c les envoie dans le même état et a fait passer de l'un à l'autre. Le seul moyen de les séparer est donc avec la transition en b. On ne pourra donc séparer 1 et 2 que si on peut séparer 4 et 5. Or, 4 et 5 envoient tous les deux au même endroit, il n'y a donc pas d'espoir de les séparer.

6 et 7, via a, envoient tous les deux en 6, qui appartient à C_β, tandis que 5 envoie en 2, qui est dans C_α. On divise donc C_β en $C_\gamma = \{4, 5\}$ et $C_\zeta = \{6, 7\}$. Ne reste plus qu'à vérifier si 6 et 7 sont séparables, ce qui est le cas via b (7 envoie dans C_ζ, 6 dans C_γ).

Au final, on a 5 classes : $C_T = \{3\}$, $C_\alpha = \{1, 2\}$, $C_\gamma = \{4, 5\}$, $C_6 = \{6\}$ et $C_7 = \{7\}$. On peut donc donner l'automate minimal suivant :



Question 2 Donnez le langage reconnu par l'automate.

Correction On pourrait appliquer littéralement l'algorithme de McNaughton et Yamada, mais on va essayer de le faire de façon un peu maligne. Tout parcourt réussi dans l'automate se divise de la façon suivante :

- Boucler entre (1,2) et (4,5), puis aller en 3
  - $(1, 2) \to (4, 5) \to^* (1, 2) = b^+a$
  - Cette boucle est donc $(a + b^+a)^* = (b^*a)^*$
  - + aller en 3 : $(b^*a)^*c$
- Passer de 3 à 3 (faire une "grande boucle") autant de fois qu'on veut
  - D'abord il faut aller de 3 à 6, éventuellement en passant par 7, soit $a + bb^*a = a + b^+a = b^*a$
  - Ensuite, boucler sur 6 puis aller en (4,5) : a^*b

- Boucler directement[1] sur (4,5) puis aller en (1,2) : b^*a
- Boucler entre (1,2) et (4,5), puis aller en c : $(b^*a)^*c$
- En mettant tout ça bout à bout, on obtient $b^*aa^*bb^*a(b^*a)^*c$. Vu qu'on peut le faire autant de fois qu'on veut, c'est partie est $(b^*aa^*bb^*a(b^*a)^*c)^*$

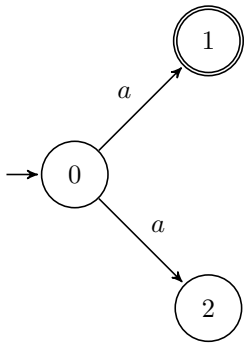
En collant les deux morceaux, on obtient $(b^*a)^*c(b^*aa^*bb^*a(b^*a)^*c)^*$

Complémentaire

Rappel Pour déterminiser un automate, il faut

1. Le déterminiser s'il ne l'est pas
2. Le compléter s'il ne l'est pas
   - Ca inclut le fait de mettre des boucles sur l'état poubelle !
3. Transformer les états terminaux en états non-terminaux et inversement
   - L'état poubelle est lui aussi concerné !

Pour vous en convaincre, je vous invite à essayer de donner le complémentaire de l'automate suivant en sautant volontairement une des deux premières étapes :

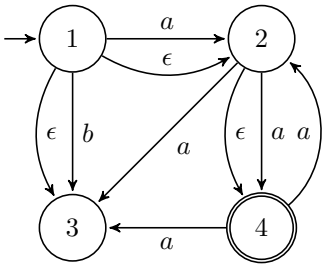


Indice Si vous oubliez de le déterminiser, le complémentaire acceptera le mot a, qui était déjà accepté par l'automate initial et ne devrait donc pas être reconnu par le complémentaire. Si vous oubliez de rajouter un état poubelle ou de le passer en terminal, vous n'accepterez pas aa, qui n'est pas reconnu par l'automate initial et devrait donc être reconnu par le complémentaire. Si vous oubliez les boucles sur la poubelle, idem avec aaa.

Elimination d'ϵ-transitions

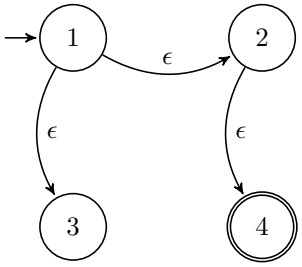
Soit l'automate suivant :

[1] Les aller-retours entre (1,2) et (4,5) sont traités dans l'étape suivante, pas la peine de se répéter. Il faut cependant bien prendre en compte la boucle en b sur (4,5) car elle peut arriver avant le premier aller-retour



Question Éliminez les ϵ-transitions.

Correction On regarde d'abord comment on peut passer gratuitement d'un état à l'autre :

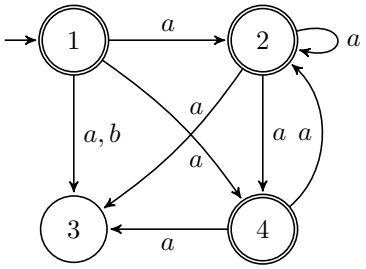


On voit que

- 1 doit "récupérer" toutes les transitions de
  - 3 (coup de bol, il n'y en a pas)
  - 2, c'est à dire aller vers 3 et 4 en a
  - 4, c'est à dire aller vers 2 et 3 en a
- 2 doit "récupérer" toutes les transitions de 4, c'est à dire aller vers 2 et 3 en a
  - Attention à ne pas oublier d'enrichir les états qui n'ont pas l'air important parce qu'ils sont au milieu d'une chaîne d'ϵ-transitions, comme 2 ici !

De plus, puisque 4 est terminal, tous les états qui y accèdent gratuitement doivent l'être également (puisqu'on devrait pouvoir "valider" la lecture d'un mot depuis ces états). 1 et 2 deviennent donc terminaux.

On obtient au final :



Automate → regex

Soit la regex suivante : $c(ab)^*ac + aac(ab)^*$

Question 1 En utilisant la méthode de Thompson, donnez un automate reconnaissant le même langage

Correction On identifie les morceaux de regex facile à transformer en automate qu'on va pouvoir combiner :

Tout d'abord, les automates qui reconnaissent c, ab, ac et aac :

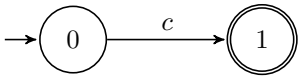


Figure 1: Automate reconnaissant c

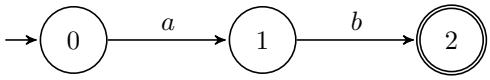


Figure 2: Automate reconnaissant ab

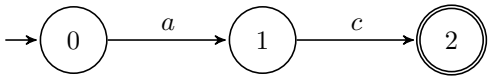


Figure 3: Automate reconnaissant ac

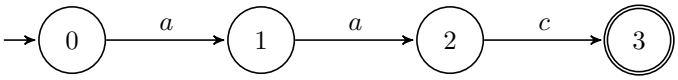


Figure 4: Automate reconnaissant aac

On construit un automate reconnaissant $(ab)^*$ à partir du deuxième automate, en ajoutant un état initial et terminal, qui a une ϵ-transition vers les états (anciennement) initiaux, et vers lequel les états (anciennement) terminaux en ont une :

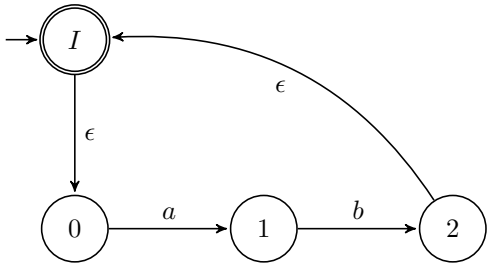


Figure 5: Automate reconnaissant $(ab)^*$

On peut alors construire un automate reconnaissant $c(ab)^*ac$ en combinant ceux de c, $(ab)^*$ et ac. Pour rappel, pour concaténer des automates, les états (anciennement) terminaux du premier envoient sur les états (anciennement) initiaux du second avec une ϵ-transition :

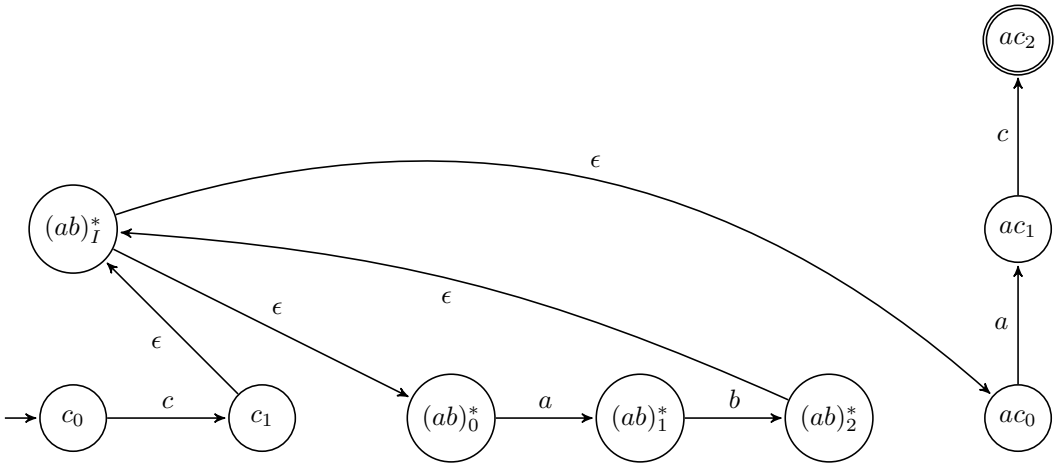


Figure 6: Automate reconnaissant $c(ab)^*ac$

De même, on reconnaît $aac(ab)^*$ avec

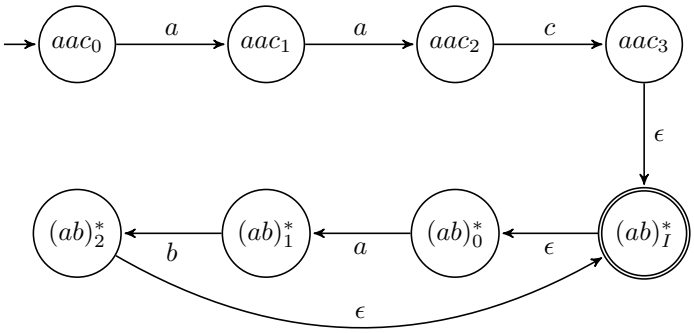


Figure 7: Automate reconnaissant $aac(ab)^*$

L'automate reconnaissant l'union des deux expressions est tout simplement l'union des automates, cad

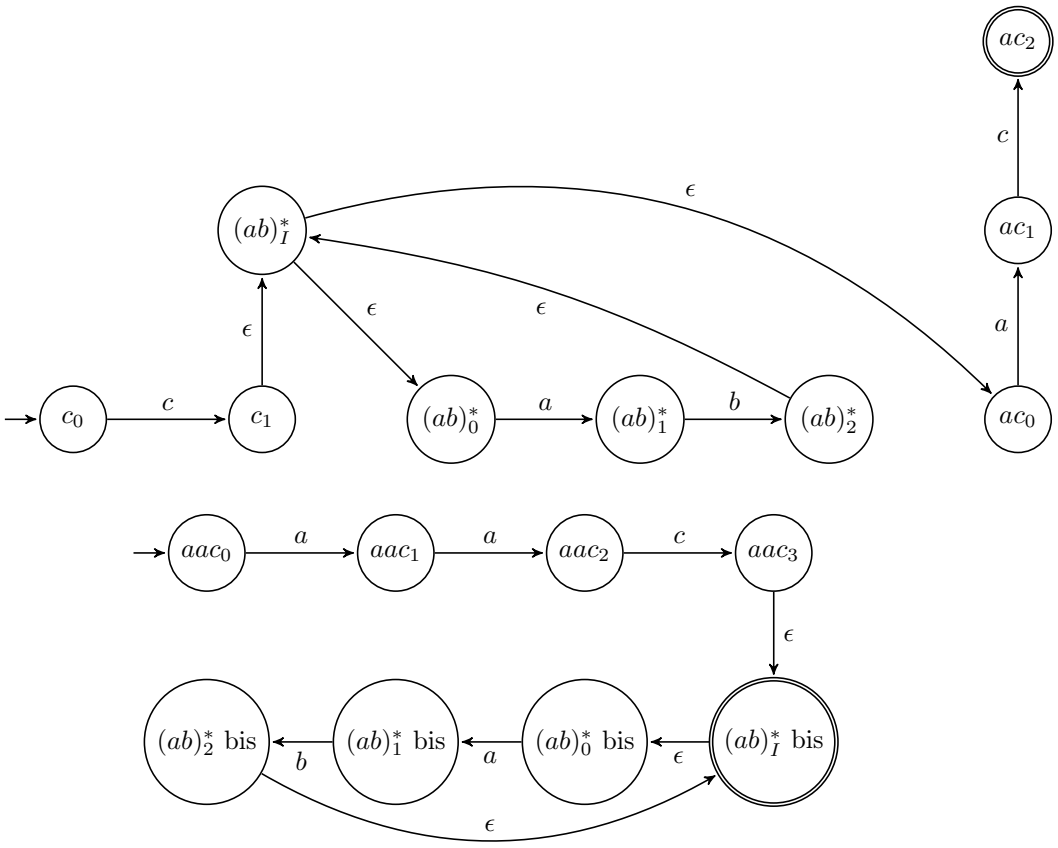


Figure 8: Automate reconnaissant $c(ab)^*ac + aac(ab)^*$

Remarque Notez que les états partagés par les deux automates ont été renommés (cf. les "bis") pour éviter toute confusion.

Question 2 En utilisant la méthode des résiduels, donnez un automate reconnaissant le même langage

Correction On a (sur la page suivante) :

- $a^{-1}(c(ab)^*ac + aac(ab)^*) = a^{-1}c(ab)^*ac + a^{-1}aac(ab)^* = a^{-1}aac(ab)^* = ac(ab)^*$
  - $a^{-1}ac(ab)^* = c(ab)^*$
    - $a^{-1}c(ab)^* = b^{-1}c(ab)^* = \emptyset$
    - $c^{-1}c(ab)^* = (ab)^*$
      - $a^{-1}(ab)^* = b(ab)^*$
        $a^{-1}b(ab)^* = c^{-1}b(ab)^* = \emptyset$
        $b^{-1}b(ab)^* = (ab)^*$
      - $b^{-1}(ab)^* = c^{-1}(ab)^* = \emptyset$
  - $b^{-1}ac(ab)^* = c^{-1}ac(ab)^* = \emptyset$
- $b^{-1}(c(ab)^*ac + aac(ab)^*) = \emptyset$
- $c^{-1}(c(ab)^*ac + aac(ab)^*) = c^{-1}c(ab)^*ac + c^{-1}aac(ab)^* = c^{-1}c(ab)^*ac = (ab)^*ac$
  - $a^{-1}(ab)^*ac = b(ab)^*ac + c$
    - $a^{-1}(b(ab)^*ac + c) = \emptyset$
    - $b^{-1}(b(ab)^*ac + c) = (ab)^*ac$
    - $c^{-1}(b(ab)^*ac + c) = \epsilon$
  - $b^{-1}(ab)^*ac = c^{-1}(ab)^*ac = \emptyset$

Version automate, avec $e = c(ab)^*ac + aac(ab)^*$

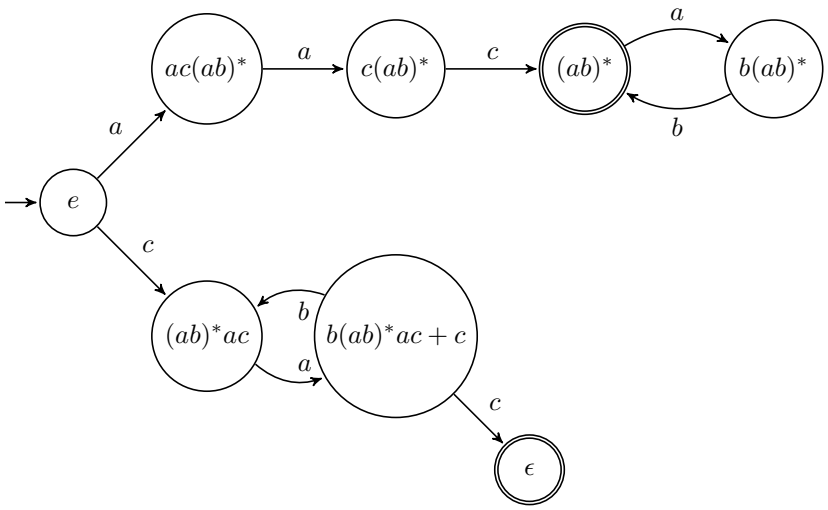


Figure 9: Automate reconnaissant $c(ab)^*ac + aac(ab)^*$