Examen de Bases formelles du TAL

Pierre-Léo Bégay

14 mai 2018

Les bonus, moins rentables, sont à garder pour la fin. De façon générale, n'hésitez pas à optimiser votre temps, notamment en ne faisant pas les exercices dans l'ordre et en sautant momentanément des questions (en laissant de place pour y revenir)

1 Algorithmique des expressions régulières [5 points]

Soient les expressions suivantes :

$e_1 = a(bb + \epsilon)a$

$e_2 = a^*(ba^*)^*$

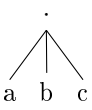
$e_3 = b(aa)^*(bb + \epsilon)$

$e_4 = bab + (abb)^*(ba + \epsilon)$

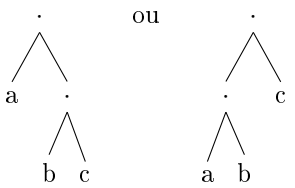
Question 1 [0,5 point]

Dressez l'arbre syntaxique de chacune de ces expressions

Note la concaténation d'expressions rationnelles étant associative, on s'autorisera à utiliser n'importe quel nombre de descendants à un $\cdot$. Concrètement, une expression comme abc pourra être représentée comme

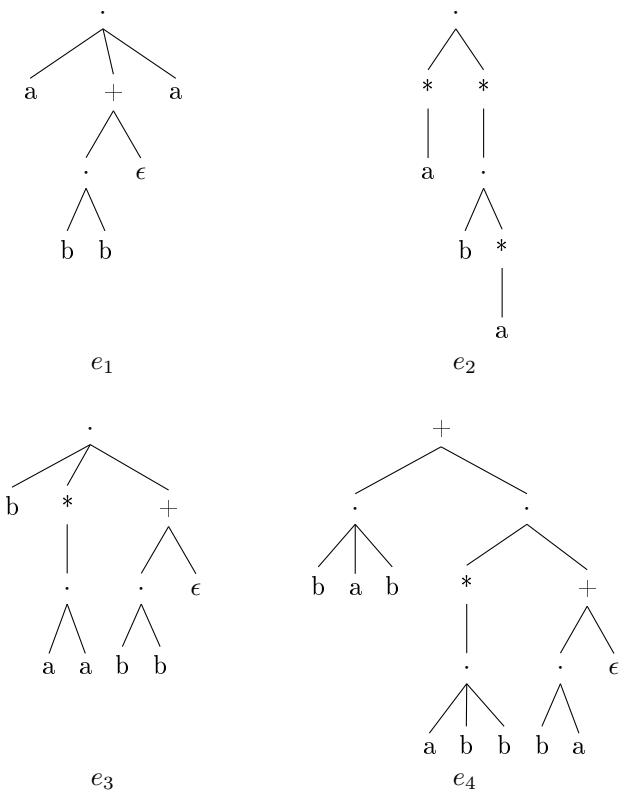


au lieu de



Note Ne perdez pas votre temps à faire des arbres propres, tant que c'est lisible ça ira.

Correction



Question 2 [1 point]

Lesquelles contiennent ϵ dans leur langage ? Pas de justification demandée.

Correction

- $e_1 = a(bb + \epsilon)a$

  Non Tout mot appartenant au langage décrit par cette expression commence (et termine) forcément par a, et a donc une longueur d'au moins 1. ϵ, dont la longueur est 0, ne fait donc pas partie du langage.

- $e_2 = a^*(ba^*)^*$

  Oui Si on choisit de remplacer toutes les * par 0, on obtient bien ϵ, qui fait donc partie du langage.

Remarque L'étoile qui est directement au-dessus du deuxième a peut en fait être rempacée par n'importe quoi, puisque l'étoile encore au-dessus, une fois remplacée par 0, *annulera* la partie ba^*.

- $e_3 = b(aa)^*(bb + \epsilon)$

  Non Tout mot appartenant au langage décrit par cette expression commence forcément par b.

- $e_4 = bab + (abb)^*(ba + \epsilon)$

Oui On a le choix d'utiliser deux sous-expressions : bab et $(abb)^*(ba + \epsilon)$. Si on choisit la seconde, on a le choix du nombre d'itérations de abb puis entre ba et ϵ. On prend 0 dans le premier cas, puis on choisit ϵ dans le second cas. On obtient donc $\epsilon\epsilon = \epsilon$.

Question 3 [3,5 points]

On veut écrire une fonction E qui, étant donnée une expression rationnelle e, renvoie *true* si le langage qu'elle décrit contient ϵ, et $false$ sinon. Pour ça, on veut utiliser un algorithme récursif, qui regarde la racine de l'arbre syntaxique de e et prend une décision en fonction de ce qu'il y trouve.

On a donc 5 cas à définir. Les 2 cas terminaux sont ceux où e est ϵ ou une lettre quelconque. Les 3 cas récursifs sont ceux sont e est composée de 2 sous-expressions unies par un $+$ ou un $\cdot$, ou e est composée d'une seule sous-expression qui est *sous* une étoile de Kleene $*$.

Le squelette d'algorithme ci-dessous sépare donc ces 5 cas[1], à vous de compléter chaque *return* **(sur votre copie ou directement sur le sujet)**. Notez que dans les cas récursifs, vous pouvez utiliser les sous-expressions.

[1] La présentation ici est proche du *pattern matching* qu'on trouverait dans un langage comme Ocaml, mais on peut coder quelque chose de similaire en python avec la primitive *isinstance(objet,classe)*

Correction

Data: Une expression régulière e
Result: Un booléen : *true* si $[\![e]\!]$ contient ϵ, *false* sinon
if $e \equiv \epsilon$ **then**

return True

else if $e \equiv a$ *(pour n'importe quel* $a \in \Sigma$*)* **then**

Une lettre n'est pas le mot vide (rappel : $\epsilon \notin \Sigma$)
return False

Si e est de la forme $e_1 + e_2$
e_1 et e_2 sont *utilisables*. On pourrait par exemple faire *return* $E(e_1)$
else if $e \equiv e_1 + e_2$ **then**

rappel : $e_1 + e_2 \approx$ les mots de e_1 + les mots de e_2
si ϵ est d'un côté **ou** de l'autre, alors il est dans l'union des deux
return $E(e_1)$ *or* $E(e_2)$

else if $e \equiv e_1 \cdot e_2$ **then**

rappel : $e_1 \cdot e_2 =$ toutes les combinaisons un mot de e_1 puis un mot de e_2
on obtient donc ϵ ssi on peut avoir $\epsilon \cdot \epsilon$
return $E(e_1)$ *and* $E(e_2)$

else if $e \equiv e_1^*$ **then**

rappel : e^* = e 'autant de fois qu'on veut', y compris 0, et $\forall e.e^0 = \epsilon$
return True

Algorithm 1: La fonction E(e), qui évalue la présence de ϵ dans $[\![e]\!]$

2 Algo des grammaires algébriques [6 points]

On rappelle qu'un symbole non-terminal A est dit **sans contribution** s'il n'existe pas de dérivation $A \rightarrow^* u$ avec $u \in \Sigma^*$, c'est-à-dire si aucune dérivation partant de A *arrive* sur un mot composé uniquement de symboles terminaux.

Question 1 [2 points]

Soit $G = \langle \{a, b\}, \{S, A, B, C, D\}, S, \{$

$S \rightarrow AA \mid BB$

$A \rightarrow CaC \mid aca$

$B \rightarrow AB \mid BA \mid CD \mid DC$

$C \rightarrow AcA \mid cac$

$D \rightarrow dCCB\}\rangle$

Donnez l'ensemble des symboles sans contribution de G (pas de justification demandée)

Correction

En utilisant l'algorithme vu en cours, on a

$N_0 = \{A, C\}$

- A est ajouté pour la règle $A \to aca$
- C est ajouté pour la règle $C \to cac$

$N_1 = \{A, C, S\}$

- S est ajouté pour la règle $S \to AA$ (puisque $AA \in (\Sigma \cup N_0)^*$)

$N_2 = \{A, C, S\}$

- B et D **ne** sont **pas** ajoutés car toutes leurs règles contiennent B ou D à droite, les parties droites ne sont donc pas dans $(\Sigma \cup N_1)^*$

Le point fixe étant atteint, on constate que les non-terminaux avec production sont A, C et S. Les non-terminaux sans production sont donc le complémentaire, à savoir B et D.

Question 2 [4 points]

On propose l'algorithme suivant pour calculer l'ensemble des non-terminaux sans contribution d'une grammaire :

Data: Une grammaire algébrique (Type 2) $G = \langle \Sigma, V, S, R \rangle$
Result: L'ensemble des non-terminaux sans contribution de V
N_0 contient l'ensemble des non-terminaux qui peuvent immédiatement se
réécrire en mot composé uniquement de terminaux
$N_0 := \{A \in V \mid A \to \alpha, \alpha \in \Sigma^*\}$
$i := 0$
repeat
  $i := i + 1$
  $N_i := N_{i-1}$ # on met tout N_{i-1} dans N_i déjà
  # On parcourt l'ensemble des règles de dérivation
  for $(A \to u) \in R$, *avec* $A \in V$ *et* u $(V \cup \Sigma)^*$ **do**
    #On parcourt l'ensemble des non-terminaux apparaissant dans la partie
    #droite de la règle
    for B *(non-terminal) apparaissant dans* u **do**
      if $B \in N_{i-1}$ **then**
        ajouter A à N_i
      end
    end
  end
until $N_i = N_{i-1}$;
return N_i

Algorithm 2: (Mauvais) calcul des symboles sans contribution de la grammaire G

Cet algorithme est incorrect, dans le sens où il ne réalise pas sa spécification. Donnez un contre-exemple concret (c'est-à-dire une grammaire ainsi qu'un symbole sans contribution pas renvoyé par l'algorithme, ou à l'inverse un symbole avec contribution qui fera partie de l'ensemble renvoyé), et expliquez comment le corriger.

Correction

Contre-exemple de l'algorithme Si on applique cet algorithme sur la grammaire de la question précédente, on obtient que tous les non-terminaux sont productifs, ce qui est faux. En effet, ça donne

$N_0 = \{A, C\}$

- A est ajouté pour la règle $A \to aca$
- C est ajouté pour la règle $C \to cac$

$N_1 = \{A, C, S, B, D\}$

- S est ajouté pour la règle $S \to AA$
- B est ajouté pour les règles $B \to AB$, $B \to BA$, $B \to CD$ et $B \to DC$
- D est ajouté par $D \to dCCB$

$N_2 = N_1$, car il n'y a rien à ajouter de toute façon

Explications Le problème de l'algorithme est qu'il estime qu'un non-terminal est productif à partir du moment où il y a **au moins un** non-terminal productif à droite d'une de ses règles, alors que l'algorithme original impose l'existence d'une règle dont **tous les** non-terminaux à droite sont productifs.

Le plus simple pour corriger l'algorithme ci-dessus est d'utiliser un *flag* 'que des prods à droite' pour chaque règle, initialisé à $True$, et mis à $False$ dès qu'on croise un non-terminal n'appartenant pas à N_{i-1}. Une fois la boucle sur la partie droite de la fonction terminée, on ajoute A à N_i uniquement si le *flag* est toujours $True$.

Bonus Expliquez en une phrase comment calculer les états sans contribution d'un automate fini, et comparez au 'bug' de l'algorithme précédent.

Brouillon de réponse

A première vue, les quantifications marchent à l'envers dans le cas des automates : pour vérifier qu'un état est productif, on veut qu'**au moins** un de ses successeurs atteigne un état terminal, pas forcément **tous**. Or, il est étrange de voir un renversement entre les grammaires et les automates, puisqu'ils sont censés se correspondre très fortement.

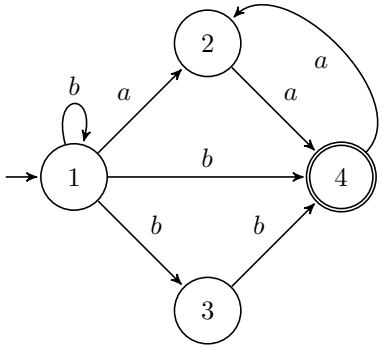
L'*erreur* ici vient du fait qu'avoir le choix entre plusieurs transitions dans un automate correspond dans les grammaires au fait d'avoir plusieurs règles avec le même non-terminal à gauche, pas à plusieurs non-terminaux à droite d'une même règle. Or, même si ça n'apparaît pas très clairement dans l'algorithme, la multiplicité des règles est traitée comme une disjonction, donc un $\exists$.

La boucle sur les non-terminaux à droite d'une règle n'a en fait pas d'équivalent dans les automates, et à raison : ces derniers correspondent aux grammaires de type 3 (l'algorithme ci-dessus est pour celles de type 2), où un seul non-terminal à droite est de toute façon autorisé. Il n'y a donc pas de différence entre $\exists$ et $\forall$.

3 Automates [6 points]

Question 1 [4 points]

Décrivez, à l'aide d'une expression rationnelle, le langage reconnu par l'automate suivant. Justifiez votre réponse, soit à l'aide de l'application d'un algorithme, soit en décrivant l'ensemble des chemins possibles (comme vu en cours).



Correction

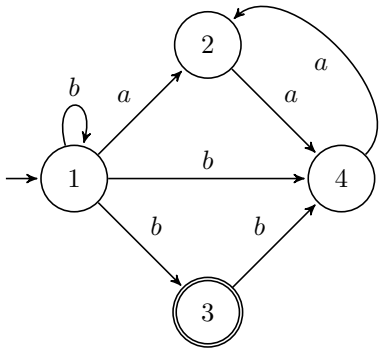
Tout parcourt dans cet automate se divise en 3 étapes :

- Au début, on peut boucler sur l'état initial 1 en lisant autant de b qu'on veut.
  - L'expression rationnelle associée est b^*
- Ensuite, on peut aller en 4 en passant par 2, par 3 ou directement
  - Les expression rationnelles associées sont, respectivement, aa, bb et b.
    L'expression rationnelle de cette étape du parcours est donc $aa + bb + b$
- Une fois qu'on a atteint l'état 4, on peut boucler autant qu'on veut entre 4 et 2, en lisant aa à chaque tour.
  - L'expression rationnelle associée est donc $(aa)^*$

En mettant toutes les étapes bout à bout, on obtient $\mathbf{b^*(aa+bb+b)(aa)^*}$

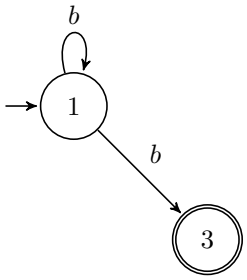
Question 2 [2 points]

Même question avec cet automate :



Correction

Il est impossible, en partant des états 2 et 4, d'atteindre 3, le seul état terminal. Ce sont des des puits, assimilables aux symboles sans production dans les grammaires. On peut donc les supprimer sans modifier le langage reconnu, ce qui donne



dont le langage reconnu est clairement $b^*b = b^+$

4 Grammaire mystère [3 points + 1 en bonus]

Soit $G = \langle \{a, b\}, \{A, B, C, D\}, A, \{$

$A \to aB \mid bD$

$B \to bC \mid aA \mid \epsilon$

$C \to aD \mid bB$

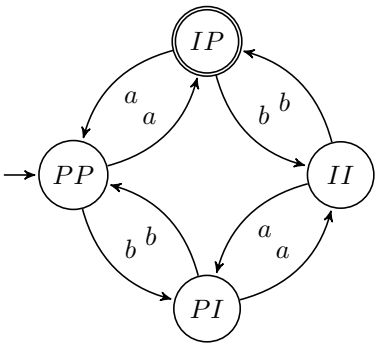
$D \to bA \mid aC\}\rangle$

Décrivez, en langue naturelle, le langage engendré par G.

Indice Vous pouvez vous appuyer sur un exemple vu en cours

Correction

L'astuce était de transformer la grammaire en automate, ce qui, après avoir remplacé A par PP, B par IP, C par II et D par PI :



On reconnaît le genre d'automate, déjà vu en cours, qui analyse la parité du nombre de a et de b dans les mots lus. PP correspond (a) pair / (b) pair, PI à pair / impair etc ... Le langage reconnu est donc clairement l'ensemble des mots avec un nombre pair de b et impair de a.

Question bonus [1 point] Donnez une expression régulière décrivant le langage en question

(non-)Correction

$((aa + bb)^*((ab + ba)(aa + bb)^*(ab + ba))^*)^*(a + (ab + ba)(aa + bb)^*b)$

A vérifier, mais ça m'a l'air pas mal