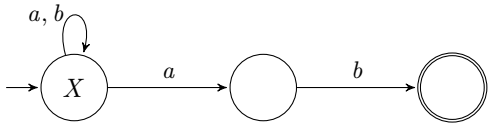
Langages (non)réguliers et applications des automates finis

Feuille « 06 » (du 11 octobre 2024)

Nous avons vu que la classe des langages réguliers correspond à la classe des langages qui sont :

— engendrés par une expression régulière, comme « $(a+b)^*(ab)$ » ;

— ou, par un système d'(in)équations régulier sur les langages, comme « $X = a.X + b.X + ab$ » ;

— ou, par un automate fini, comme 

Et, nous avons aussi vu que si L, L_1 et L_2 sont réguliers, alors L_1+L_2 , $L_1.L_2$, $L_1\cap L_2$, L^* et $\overline{L}$ sont réguliers ; de même $L_1\setminus L_2 = L_1 \cap \overline{L_2}$ est régulier.

Nous complétons ci-dessous ces propriétés des langages réguliers. Ceci donnera aussi des techniques pour prouver qu'un langage donné *n'est pas régulier*. Concernant les applications des langages réguliers, la feuille « 05 » en a illustré une très fréquente : *décrire certaines constructions syntaxiques des langages de programmation*, comme les littéraux numériques de Python. Les limites des langages réguliers étudiées ici (notamment à l'exo 10) permettront de se convaincre qu'ils ne sont pas suffisants pour représenter *intégralement* la syntaxe des langages de programmation. Nous laisserons l'étude de l'usage des langages réguliers pour représenter des « fragments » des langages de programmation pour les périodes 2 et 3. En guise de conclusion du cours, la section 4 illustre le fait que les automates finis et leurs variantes ont de nombreuses autres applications, notamment en *sémantique des programmes informatiques*.

1 Décidabilité des comparaisons entre langages réguliers

Exercice 1. Donner un algorithme qui étant donné deux expressions régulières E_1 et E_2 retourne un booléen indiquant si $\mathcal{L}(E_1) = \mathcal{L}(E_2)$.
Illustrer cet algo sur $V = \{0,1\}$ avec $E_1 = (0+\varepsilon)(10)^*(1+\varepsilon)$ et $E_2 = (1+\varepsilon)(01)^*(0+\varepsilon)$.

Exercice 2. Donner un algorithme qui étant donné deux expressions régulières E_1 et E_2 retourne un booléen indiquant si $\mathcal{L}(E_1) \subseteq \mathcal{L}(E_2)$.
Illustrer cet algo sur $V = \{a,b\}$ avec $E_1 = (a+b)^*ab$ et $E_2 = b^*a^+b(a+b)^*$.

2 Constructions préservant ou pas les langages réguliers

Dans cette section, nous admettons que le langage $M_0 \stackrel{def}{=} \{a^n b^n \mid n \in \mathbb{N}\}$ (sur le vocabulaire $\{a,b\}$) **n'est pas régulier**. Ce résultat sera montré à l'exercice 11 (sans utiliser les résultats de cette section, bien sûr).

Avertissement sur l'inclusion Dans un contexte où on sait que $L_1 \subseteq L_2$ avec un des deux langages qui est (non)régulier, on ne peut en général, rien déduire sur l'autre (cf. exo 4). Au mieux, on a une *indication*. Par exemple, si L_1 est régulier, pour montrer que L_2 régulier, on peut chercher à construire L_2 comme une *union* de langages réguliers contenant L_1 ; c'est ce qu'il faut faire à l'exo 3 pour montrer « M_2 régulier $\Rightarrow$ M_0 régulier » après avoir remarqué que $M_2 \subseteq M_0$. Symétriquement, si L_2 est régulier, pour montrer que L_1 régulier, on peut chercher à construire L_1 comme une *intersection* de langages réguliers à partir de L_2 ; c'est ce qu'il faut faire à l'exo 3 pour montrer « M_3 régulier $\Rightarrow$ M_0 régulier » après avoir remarqué que $M_0 \subseteq M_3$.

Exercice 3. Démontrer que les langages suivants ne sont pas réguliers.

$$M_2 \stackrel{def}{=} \{a^{2n}b^{2n} \mid n \in \mathbb{N}\} \qquad M_3 \stackrel{def}{=} \{w \mid w \in \{a,b\}^* \wedge |w|_a = |w|_b\}$$

$$M_1 \stackrel{def}{=} \{a^n b^m \mid n \in \mathbb{N} \wedge m \in \mathbb{N} \wedge n \neq m\}$$

Indication Faire des raisonnements par l'absurde comme ci-dessous en introduisant des transformations f qui préservent les langages réguliers :

« *Supposons M_2 régulier. Or $M_0 = f(M_2)$, donc M_2 régulier. Absurde.* »

Exercice 4. Montrer que **les deux affirmations suivantes sont fausses** à l'aide de contre-exemples

1. « *Soient L_1 et L_2 deux langages (sur un vocabulaire V fixé). Si L_1 est régulier et $L_1 \subseteq L_2$ alors L_2 est régulier.* »
2. « *Soient L_1 et L_2 deux langages (sur un vocabulaire V fixé). Si L_2 est régulier et $L_1 \subseteq L_2$ alors L_1 est régulier.* »

Exercice 5. Soient $(L_i)_{i\in\mathbb{N}}$ une famille (infinie) de langages **réguliers** sur un vocabulaire fixé V. Montrer que :

1. Pour tout $n \in \mathbb{N}$, les langages $\bigcup_{i=0}^{n} L_i$ et $\bigcap_{i=0}^{n} L_i$ sont réguliers.
2. Il **est faux** de conclure « $\bigcup_{i\in\mathbb{N}} L_i$ *est régulier* ».
3. Il **est faux** de conclure « $\bigcap_{i\in\mathbb{N}} L_i$ *est régulier* ».

NB : le « passage » aux « infinis » ne préserve généralement pas les langages réguliers !

Substitutions sur les langages Soient V_1 et V_2 deux vocabulaires fixés et soit $\sigma \in V_1 \to \mathcal{P}(V_2^*)$. À partir de σ, on définit une substitution sur les mots $\sigma^* \in V_1^* \to \mathcal{P}(V_2^*)$ où $\sigma^*(w)$ est défini par récurrence sur la taille de w, avec :

— $\sigma^*(\varepsilon) \stackrel{def}{=} \{\varepsilon\}$

— $\sigma^*(a.w) \stackrel{def}{=} \sigma(a).\sigma^*(w)$

Puis, on définit une substitution sur les langages $\sigma^\cup \in \mathcal{P}(V_1^*) \to \mathcal{P}(V_2^*)$ avec

$$\sigma^\cup(L) \stackrel{def}{=} \bigcup_{w\in L} \sigma^*(w)$$

On dit que σ est une substitution régulière ssi pour tout $a \in V_1$, le langage $\sigma(a)$ est régulier.

Exercice 6. (avancé) Soit $\sigma \in V_1 \to \mathcal{P}(V_2^*)$ quelconque *pas forcément régulière* !
Montrer les propriétés suivantes sur $\sigma^\cup$ (pour tous langages L, L_1 et L_2 sur le vocabulaire V_1) :

1. $\sigma^\cup(L_1 \cup L_2) = \sigma^\cup(L_1) \cup \sigma^\cup(L_2)$;
2. $\sigma^\cup(L_1.L_2) = \sigma^\cup(L_1).\sigma^\cup(L_2)$;
3. $\sigma^\cup(L^*) = (\sigma^\cup(L))^*$.

Abus de notation Dans la suite, on écrit abusivement « σ » au lieu de « σ^* » ou de « $\sigma^\cup$ ».

Exercice 7. Trouver une substitution régulière σ telle que $\sigma(abac^*) = (aaa)^*(bbb)^*$.

Exercice 8. Soit $E \in \mathbb{E}(V_1)$ et soit $\sigma \in V_1 \to \mathcal{P}(V_2^*)$ une substitution régulière. Montrer par induction sur E que $\sigma(\mathcal{L}(E))$ est un langage régulier.
On en conclut que si L est un langage régulier sur V_1 alors $\sigma(L)$ est régulier sur V_2.

Exercice 9. Soit $M_2' \stackrel{def}{=} \{b^{2n}a^{2n} \mid n \in \mathbb{N}\}$. Montrer que M_2' n'est pas régulier.

Exercice 10. Soit V un vocabulaire contenant au moins le sous-ensemble de 2 symboles $\{(,)\}$. Soit L un langage sur V contenant un mot $w_0 \in L$ et qui satisfait les 3 propriétés suivantes :

1. pour tout $w \in L$, on a $|w|_(= |w|_)$;
2. $|w_0|_(= 0$;
3. pour tout $w \in L$, on a $(.w.) \in L$.

Remarque : on dit d'un tel langage L qu'il incorpore un système de parenthésage (c'est le cas par exemple, des langages d'expressions). **Montrer que L n'est pas régulier.**

3 Lemme de l'étoile

Le lemme de l'étoile ci-dessous exprime une propriété *nécessairement* vérifiée par tout langage régulier. Il sert en fait à montrer qu'un langage L *n'est pas régulier* en montrant que L ne satisfait pas la propriété. Intuitivement, le lemme dit que si L est un langage régulier et infini, alors tout mot z « suffisamment » grand est « engendré » par « itération » d'au moins une « $*$ » (cf. « Théorème des langages finis » à la feuille « 03 »). Autrement dit, z appartient à un sous-langage *inclus* dans L qui est de la forme « $u.v^*.w$ » où « $z = u.v.w$ ». *In fine*, si L n'a pas *suffisamment* de sous-langages de cette forme, alors il n'est pas *suffisamment* « régulier » (au sens informel) et n'est pas *régulier* (au sens de la théorie des langages formels).

Énoncé du lemme de l'étoile Soit L un langage régulier sur un vocabulaire V fixé. Alors, il existe $N \in \mathbb{N}$ tel que pour tout $z \in L$ vérifiant $|z| \geq N$, il existe trois mots u, v, w de V^* tels que $z = u.v.w$ avec

1. $|v| \geq 1$ et $|u.v| \leq N$;
2. et pour tout $i \in \mathbb{N}$, $u.v^i.w \in L$.

Exercice 11. Démontrer que le langage $M_0 \stackrel{def}{=} \{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas régulier en appliquant le lemme de l'étoile.

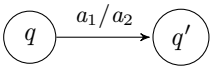
Exercice 12. (avancé) Démontrer le lemme de l'étoile.

Exercice 13. Démontrer que chacun des langages $L_1 \stackrel{def}{=} \{a^m b^n \mid m \geq n\}$ et $L_2 \stackrel{def}{=} \{a^m b^n \mid m \leq n\}$ n'est pas régulier en appliquant directement le lemme de l'étoile.

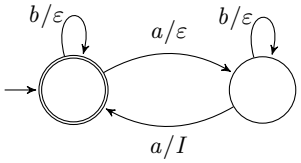
4 Extensions des automates finis pour/avec la sémantique

Nous présentons ici très informellement quelques extensions des automates. L'idée est juste d'illustrer l'omniprésence des automates dans les modèles informatiques. Voir aussi https://en.wikipedia.org/wiki/Finite-state_machine.

Transducteurs finis par machines de Mealy Les machines de Mealy sont une extension de automates finis modélisant la traduction d'un langage (régulier) sur un vocabulaire V_1 dans un langage (régulier) sur un vocabulaire V_2. A chaque transition de l'automate, en plus du symbole d'entrée dans V_1, est associé un symbole de sortie dans V_2 (éventuellement ε).
Formellement, une machine de Mealy est défini comme un 6-uplet $\langle Q, V_1, V_2, s, \delta, F\rangle$ avec $\delta \in Q \times (V_1 \cup \{\varepsilon\}) \to \mathcal{P}(Q \times (V_2 \cup \{\varepsilon\}))$, de sorte qu'en plus de *consommer* un mot d'entrée $w_1 \in V_1^*$, les chemins de s à un état de F *produisent* un mot de sortie $w_2 \in V_2^*$.

Concrètement, une transition $(q', a_2) \in \delta(q, a_1)$ est dessinée 

Par exemple, la machine de Mealy ci-contre accepte en entrée les mots w_1 de $\{a, b\}^*$ tels que $|\, w_1 \,|_a = 2n$ avec $n \in \mathbb{N}$. Le mot w_2 de sortie de $\{I\}^*$ alors associé à w_1 est I^n (c-à-d. n codé en unaire).



Exercice 14. On considère ici $V_1 = \{0, 1, \$\}$ et $V_2 = \{0, 1\}$. Définir une machine de Mealy qui reconnaît les entiers écrits en binaire, poids faibles d'abord, et produit en sortie cet entier binaire additionné de 1, le symbole « \$ » étant un marqueur de fin du mot d'entrée. Par exemple, sur l'entrée « 11010\$ » qui représente l'entier onze, la machine produira le mot « 00110 » qui représente l'entier 12 (le bit éventuellement produit par « \$ » est donc un bit de retenu). De façon générale, le langage d'entrée de la machine devra être $(0 + 1)^+\$$.

```python
def semF(w):
  r = 0
  i = 0
  while i < len(w):
    match w[i]:
      case '0':
        r *= 2
      case '1':
        r = 2*r+1
      case ',':
        i += 1
        break
      case _:
        assert False
    i += 1
  d = .5
  while i < len(w):
    match w[i]:
      case '0':
        pass
      case '1':
        r = r+d
      case _:
        assert False
    d = d/2
    i += 1
  return r
```

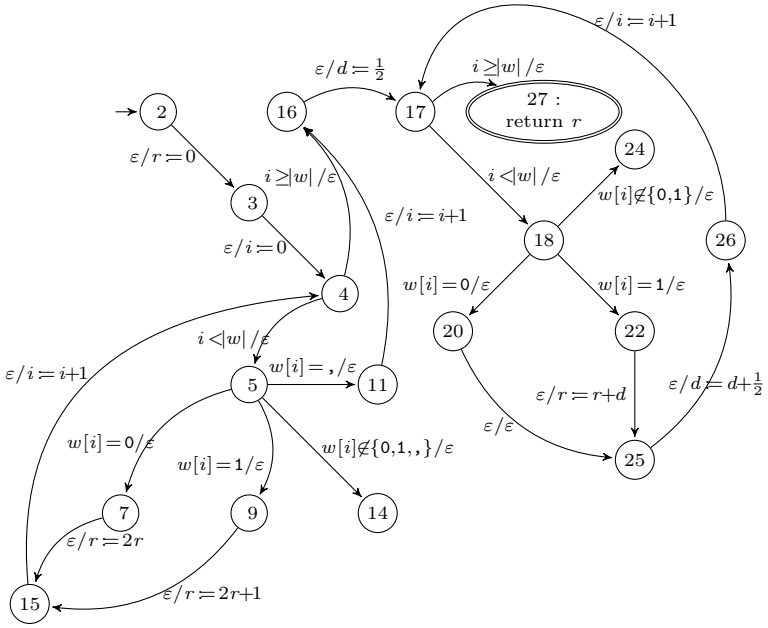
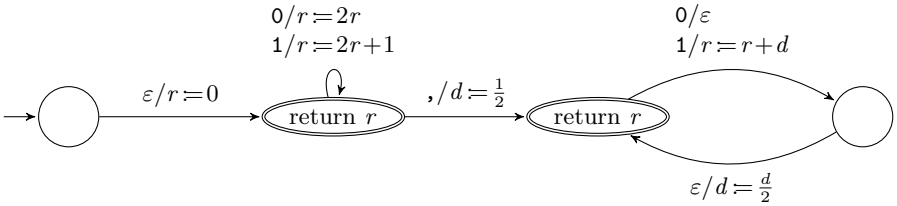


FIGURE 1 – Calcul de l'interprétation d'un nombre binaire à virgule

Automates de traitement de langages à la Mealy Une variante des machines de Mealy consiste à mettre en tant que « vocabulaire de *sortie* » des « actions sémantiques » interprétées dans le métalangage. Typiquement ces « actions » utilisent des variables sémantiques lues et modifiées pendant l'exécution de l'automate, l'action ε correspondant à « ne rien faire » comme l'instruction **pass** de PYTHON. Par exemple, l'algorithme implémenté en PYTHON en colonne gauche de la figure 1 (et solution du « RENDU 1 » de la feuille « 02 ») est décrit très simplement par la machine ci-dessous utilisant deux variables sémantiques r et d. Ici, le « return r » dans les états finaux indique que la sémantique associée au mot reconnu est uniquement donnée par r (la valeur de d peut être oubliée) :



Représentation de programmes à base d'automates de Mealy Au delà de l'algorithme, on peut représenter le programme lui-même par la machine donnée en colonne droite de la figure 1. En plus des « actions » en tant que « vocabulaire de *sortie* », cette représentation utilise des « conditions booléennes » sur les variables sémantiques en tant que « vocabulaire *d'entrée* ». Ces « conditions » déterminent ainsi quelles sont les transitions applicables : une transition n'est applicable que si sa condition est vraie ; dans ce contexte, ε représente donc la condition vraie. Chaque état de l'automate représente alors une *ligne* de code. Le formalisme des automates sert ici à décrire le *flot*

de contrôle du programme[1]. Remarquons sur la figure 1 que certaines lignes ne sont pas représentées en tant qu'état, mais uniquement en tant que transition. Par exemple, l'instruction **break** à la ligne 12 est directement représentée par la transition de la ligne 11 à la ligne 16. De même, les instructions **case** sont directement représentées par les conditions sur les transitions issues des instructions **match**. D'autres instructions donnent une certaine forme à l'automate. Par exemple, les instructions **while** correspondent à une « tête de boucle ». L'instruction **return** correspond à un *état final* sans transition sortante. Et les instructions **assert** `False` sont ici directement interprétées comme des *états puits* sans transition sortante. Dans le cas général, une instruction « **assert** c » serait juste représentée à l'aide d'une transition « c/ε ».[2]

Ce type d'automates peut en fait représenter n'importe quel corps de fonction (ou procédure) d'un programme informatique. Ceci présente l'intérêt de pouvoir s'inspirer des traitements sur les automates finis pour analyser ou transformer les programmes informatiques. Par exemple, pour vérifier (automatiquement) qu'un programme informatique satisfait une certaine propriété (simple), on peut utiliser une généralisation du test d'inclusion de langages vu à l'exo 2. On peut aussi *optimiser* (automatiquement) les programmes, en élaguant le *code inaccessible* (code mort) ou en *factorisant* les portions de code équivalentes, comme dans la minimisation d'automates. C'est ce que font les *compilateurs optimisants* sur une représentation proche de celle de la figure 1, appelée *graphe de flot de contrôle*.[3]

Exercice 15. Dessiner l'automate de Mealy représentant le flot de contrôle du corps de la fonction `syracuse_odd` ci-dessous.

```
1  def syracuse_odd(n):
2    i = 0
3    while n != 1:
4      a = n//2
5      if n == 2*a:
6        n = a
7        continue
8      i += 1
9      n = 3*a + 2
10   return i
```

Modélisation des exécutions infinies Les traces traitées par les machines de la représentation précédente sont nécessairement finies : ce sont celles qui terminent dans un état final. Pour représenter des programmes avec des exécutions potentiellement infinies (comme le logiciel de contrôle d'une centrale nucléaire, ou tout simplement l'OS de votre PC), il peut être commode d'étendre le formalisme des automates aux traces infinies. Cela motive la théorie des *ω-automates*[4] qui traitent des *mots infinis*[5]. Cette généralisation nécessite toutefois d'adapter en profondeur la théorie étudiée dans ce cours, ce que nous n'avons pas le temps de faire...

1. https://en.wikipedia.org/wiki/Control_flow
2. Ce choix correspond à une sémantique *simple* où les **assert** ne sont jamais rattrapés dans un **try** et que leur exécution correspond à une utilisation proscrite du programme. Pour prendre en compte le rattrapage d'exception, il faudait représenter le cas d'échec du **assert** dans l'automate. Par exemple, en cas de rattrapage à l'intérieur de la fonction, l'échec d'un « **assert** `c` » se traduirait par une transition « $\neg c/\varepsilon$ » vers la ligne rattrapant cette exception. Sinon, en l'absence de rattrapage à l'intérieur de la fonction, le cas d'échec du **assert** serait représenté en introduisant un état final dédié exprimant que l'exécution de la fonction termine avec une exception.
3. https://en.wikipedia.org/wiki/Control-flow_graph
4. https://fr.wikipedia.org/wiki/Automate_sur_les_mots_infinis
5. https://fr.wikipedia.org/wiki/Mot_infini