Langages formels (11X003) - Automne 2023

4. Expressions régulières

Enseignant: Arnaud Casteigts *Assistant: Alexandre-Quentin Berger*

4.1 Langages réguliers (rappel)

Rappelons que les langages réguliers sont définis inductivement comme suit :

- $\emptyset$ et $\{\varepsilon\}$ sont des langages réguliers,
- Pour tout $\mathtt{s} \in \Sigma$, $\{\mathtt{s}\}$ est un langage régulier,
- Si L est un langage régulier, alors L^* l'est aussi,
- Si L_1 et L_2 sont des langages réguliers, alors $L_1 \cup L_2$ et $L_1 \circ L_2$ le sont aussi.

Cette définition implique aussi que L^+ est régulier si L est régulier, car $L^+ = L \circ L^*$.

4.2 Expressions régulières

Les expressions régulières (ER) permettent de décrire des langages réguliers de façon textuelle. Elles sont définies comme ces derniers, de manière inductive et en utilisant les mêmes règles, mais en simplifiant les notations. Notamment, les accolades sont omises et la concaténation $\circ$ est généralement omise aussi. Par ailleurs, des parenthèses peuvent être utilisées pour gérer les priorités entre opérations, comme pour les expressions arithmétiques classiques.

Par exemple, sur l'alphabet $\Sigma = \{a, b, c\}$, l'expression $(a \cup b)^* ca^+$ désigne le langage $(\{a\} \cup \{b\})^* \circ \{c\} \circ \{a\}^+$, à savoir tous les mots qui contiennent exactement un `c`, précédé de n'importe quel préfixe sans `c` et suivi d'un nombre arbitraire (mais strictement positif) de `a`. On a donc $(a \cup b)^* ca^+ = \{ca, aca, bca, caa, aaca, abca, baca, bbca, caaa, aacaa, abcaa, \dots\}$

Outre leur interêt dans l'étude des langages formels, les expressions régulières ont de nombreuses applications, notamment pour la recherche de motif textuels dans les fichiers. Plusieurs langages de programmation et commandes UNIX les supportent nativement.

4.2.1 Quelques exemples

Voici quelques exemples d'ERs sur l'alphabet $\Sigma = \{0, 1\}$, tirés du livre de Michael Sipser (*Introduction to the Theory of Computation*, ouvrage de référence en anglais) :

- 0^*10^* = mots contenant un seul 1.
- $\Sigma^*1\Sigma^*$ = mots contenant au moins un 1.
- $\Sigma^*001\Sigma^*$ = mots contenant le facteur 001.
- $1^*(01^+)^*$ = mots dans lesquels chaque 0 est suivi d'au moins un 1.
- $(\Sigma\Sigma)^*$ = mots de longueur paire.
- $(\Sigma\Sigma\Sigma)^*$ = mots dont la longueur est multiple de trois.
- $01 \cup 10 = \{01, 10\}$.
- $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$ = mots qui commencent et finissent par le même symbole.
- $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.

Remarque : on authorise l'usage de Σ^* dans les expressions régulières, avec la signification habituelle de "n'importe quel facteur construit sur l'alphabet Σ". Ici, Σ^* est donc synonyme de $\{0 \cup 1\}^*$.

Soit L un langage régulier, on a :

- $L \circ \emptyset = \emptyset$ (rappel : $\emptyset$ est l'élément absorbant de la concaténation)
- $L \cup \emptyset = L$ (rappel : $\emptyset$ est l'élément neutre de l'union)
- $L \circ \varepsilon = L$ (rappel : ε est l'élément neutre de la concaténation)
- $L \cup \varepsilon = L$ auquel est ajouté ε s'il n'y était pas déjà.

  Il en va de même pour les expressions régulières, bien sûr.

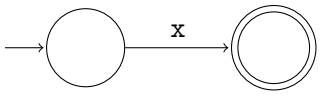
4.3 Tout langage régulier peut être reconnu par un AFN

Dans cette section, nous allons (enfin !) montrer que les automates fini peuvent reconnaître n'importe quel langage régulier. Rappelons que les deux types d'automates finis que nous connaissons (AFD et AFN) sont équivalents en termes de langages reconnus. Il suffit donc de montrer que tout langage régulier peut être reconnu par un AFN.

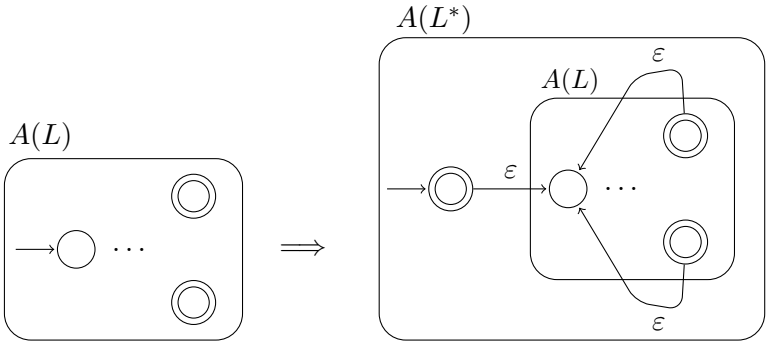
Comment faire ? L'idée est de montrer que les AFNs peuvent être combinés en suivant les mêmes opérations que celles qui définissent les langages réguliers. Nous procédons en reprenant donc, étape par étape, la définition des langages réguliers donnée plus haut.

- Les langages $\emptyset$ et $\{\varepsilon\}$ peuvent être reconnus par les automates suivants (à gauche et à droite, respectivement) :

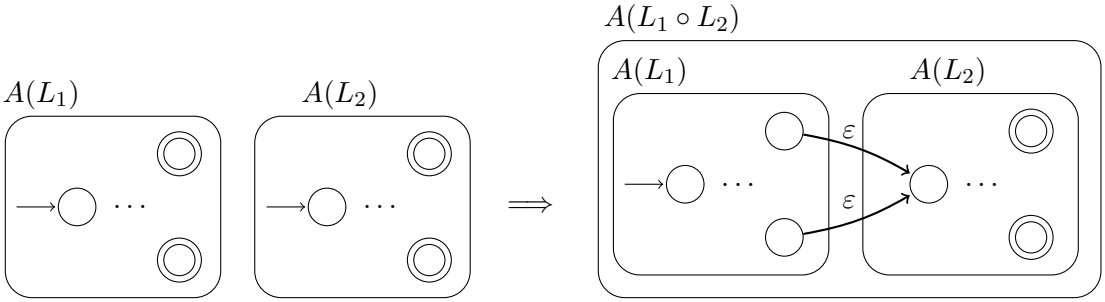
- Pour chaque symbole $\mathtt{x} \in \Sigma$, le langage $\{\mathtt{x}\}$ peut être reconnu par l'automate suivant :



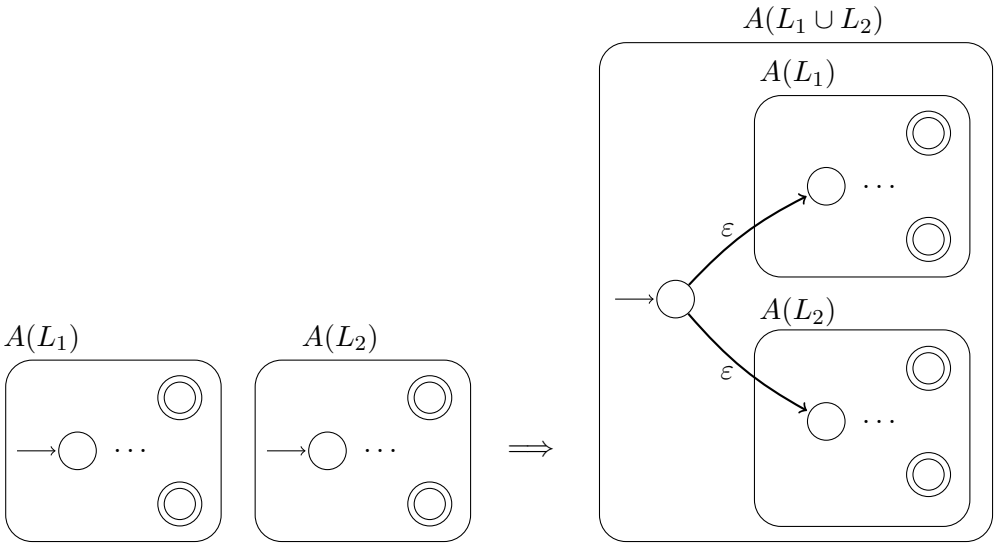
- Si L est reconnu par un AFN $A(L)$, alors L^* est reconnu par l'automate $A(L^*)$ suivant :



- Si L_1 est reconnu par un AFN $A(L_1)$ et L_2 est reconnu par un AFN $A(L_2)$, alors $L_1 \circ L_2$ est reconnu par l'automate $A(L_1 \circ L_2)$ suivant :



- Si L_1 est reconnu par un AFN $A(L_1)$ et L_2 est reconnu par un AFN $A(L_2)$, alors $L_1 \cup L_2$ est reconnu par l'automate $A(L_1 \cup L_2)$ suivant :



4.3.1 Commentaires

Pour l'opération L^*, la construction consiste à :

- Remplacer l'ancien état initial q_0 par un nouvel état initial (également final) et ajouter une ε-transition depuis cet état vers q_0.
- Ajouter une ε-transition depuis chaque ancien état final vers q_0.

Le fait que l'ancien état initial soit aussi final permet d'entrer "zéro fois" dans l'automate et donc de reconnaître L^0. Le fait que les états finaux renvoient vers q_0 permet de répéter le passage dans l'automate autant de fois qu'on veut, et donc de reconnaître L^k pour tout $k \geq 1$. On reconnaît donc bien L^*.

Pour l'opération $L_1 \circ L_2$, la construction consiste à :

- Brancher tous les états sortants de l'automate reconnaissant L_1 à l'état initial de l'automate reconnaissant L_2 (via des ε-transitions), ce qui permet de passer dans le second automate à chaque fois que le préfixe lu jusqu'à présent est dans L_1.
- Rendre non-initial l'ancien état initial de l'automate reconnaissant L_2, pour obliger à passer d'abord par L_1.
- Rendre les états finaux de L_1 non-finaux, pour obliger à passer ensuite par L_2.

Cette construction reconnaît donc exactement $L_1 \circ L_2$.

Pour l'opération $L_1 \cup L_2$, la construction consiste à :

- Remplacer les deux états initiaux par un seul nouvel état initial, qui y sera relié par des ε-transitions. Cela permet d'explorer en parallèle (indépendamment) les deux automates, donc d'accepter un mot si au moins l'un des deux automates l'aurait accepté.

Ainsi, pour tout langage régulier (et donc pour toute expression régulière le représentant), il existe un AFN qui reconnaît exactement ce langage. Et comme nous avons vu que tout AFN peut être déterminisé en AFD, cela donne la chaîne de transformation suivante :

$$\text{ER} \longrightarrow \text{AFN} \longrightarrow \text{AFD}.$$

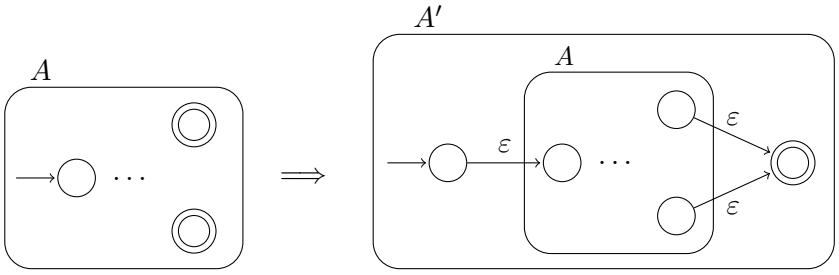
Il ne reste plus qu'à refermer le cycle en montrant que tout AFD peut être transformé en une expression régulière, et nous aurons alors montré que tous ces outils sont équivalents.

4.4 Tout AFD peut être tranformé en expression régulière

Nous allons voir ici une autre transformation qui convertit un AFD quelconque en une expression régulière qui représente exactement le même langage. La première étape consiste à transformer notre AFD A en un AFN A' qui a les propriétés suivantes :

- L'état initial de A' n'a aucune transition *entrante*.
- Il n'y a qu'un état final de A', et il n'a aucune transition *sortante*.

Cette transformation est simple, il suffit d'ajouter un nouvel état initial qui remplace l'ancien et qui le relie par une ε-transition. De même, il suffit d'ajouter un nouvel état final qui remplace tous les anciens, avec une ε-transition provenant de chacun d'entre eux. Clairement, ce nouvel automate reconnaît le même langage que A.



La deuxième étape consiste à supprimer, les uns après les autres, tous les états de A dans l'automate A', en le modifiant au fur et à mesure jusqu'à ce qu'il ne reste plus que l'état initial et l'état final, avec une seule transition entre les deux. Cette transition indiquera (magie) l'expression régulière que nous cherchons.

L'intuition est la suivante. Supposons que nous avons, quelque part dans l'automate A', trois états q, q_1 et q_2 avec, par exemple, les transitions suivantes (dessin de gauche) :

En regardant le dessin de gauche, on comprend que l'état q et ses transitions permettent, plus globalement, d'aller de q_1 à q_2 en lisant n'importe quel facteur correspondant à l'expression régulière $\texttt{ac}^*\texttt{b}$ (s'il n'y avait pas de boucle sur q, ce serait juste ab). Sur cet exemple, q ne sert à rien d'autre. On peut donc le supprimer ainsi que ses transitions, en le remplaçant par une nouvelle transition de q_1 à q_2 correspondant à l'expression $E = \texttt{ac}^*\texttt{b}$ (dessin de droite). L'automate ainsi obtenu n'est plus vraiment un AFN, car les AFN sont censés lire un seul symbole à la fois, mais il reconnaît bien le même langage.

Si vous avez compris l'opération ci-dessus, vous avez l'intuition principale (sinon, relisez le paragraphe précédent). Le cas général est un peu plus complexe, car :

1. Dans le cas général, il pourrait déjà exister une transition de q_1 à q_2.
   → Solution : si une transition entre q_1 et q_2 existe déjà avec un autre symbole, ou plus généralement avec une autre expression E', il suffit d'y ajouter la nouvelle expression en faisant l'union $E' \cup E$ (signifiant qu'on peut aller de q_1 à q_2 en lisant le motif E' ou le motif E).
2. Dans le cas général, l'état q pourrait être utilisé comme intermédiaire entre plusieurs couples d'états, pas seulement q_1 et q_2.
   → Solution : quand on veut supprimer un état q, il faut considérer *tous les couples d'états possibles* qui utilisent q comme intermédiaire, et ajouter toutes les transitions correspondantes, avant de pouvoir supprimer q.
3. Un état q_1 pourrait avoir des transitions vers q et depuis q (flèches dans les deux sens). L'état q serait donc un intermédiaire pour aller de q_1 à q_1 lui-même.
   → Solution : c'est la même chose qu'entre q_1 et q_2, sauf que la transition ajoutée (ou sur laquelle on utilise l'union) est une boucle sur q_1.

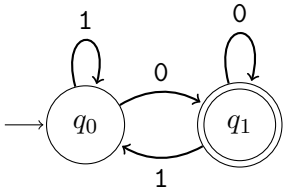
Ces éléments en tête, voici l'algorithme général : On notera ici S l'ensemble des états à supprimer, qui sont tous les états de Q'_A sauf son nouvel état initial et nouvel état final :

Tant que $S \neq \emptyset$:
| Choisir un état q de S (arbitrairement)
| Pour chaque état $q_1 \neq q$ de Q'_A :
| | Pour chaque état $q_2 \neq q$ de Q'_A :
| | | Créer l'expression E correspondant aux transitions $q_1 \to \overset{\frown}{q} \to q_2$ (si applicable)
| | | Ajouter (ou créer) E comme transition entre q_1 et q_2 (comme discuté ci-dessus)
| Enlever q de S et de $Q_{A'}$

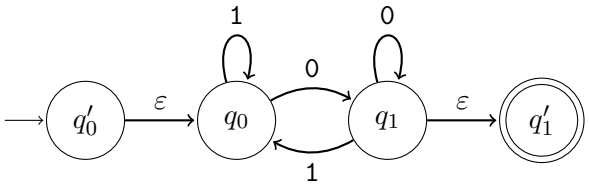
À l'issue de ces opérations, il ne reste plus que l'état initial et l'état final dans $Q_{A'}$ et la transition entre ces deux états correspond bien à une expression qui décrit le langage reconnu par A. Nous ne le démontrerons pas ici, mais cela pourrait se faire par induction sur le nombre d'états de $Q_{A'}$, en montrant qu'après chaque suppression d'état, le langage reconnu par l'automate a été préservé.

4.4.1 Exemple

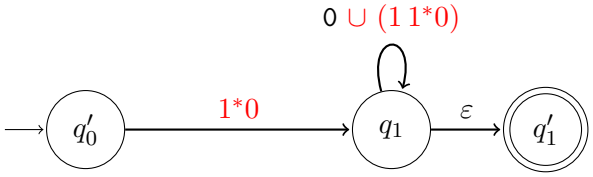
Voyons maintenant un exemple simple. Nous allons convertir l'AFD qui reconnaît les nombres pairs sur l'alphabet $\Sigma = \{0, 1\}$, c.à.d. les mots se terminant par zéro. L'expression régulière correspondante est $\Sigma^*0 = (0 \cup 1)^*0$.



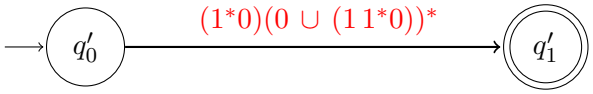
Ajout des nouveaux états (initial et final) :



Supprimons d'abord q_0 (choix arbitraire). Cet état sert d'intermédiaire de deux manières différentes : (1) de q'_0 vers q_1, avec pour expression correspondante $\varepsilon 1^*0 = 1^*0$, on ajoute donc une transition 1^*0 de q'_0 à q_1 ; (2) de q_1 vers lui-même, avec pour expression 11^*0, on ajoute donc cette expression (via une union) sur la boucle existante de q_1. L'état q_0 peut maintenant être supprimé, ce qui donne :



Supprimons maintenant q_1. Cet état sert d'intermédiaire d'une seule manière, de q'_0 à q'_1 avec l'expression $(1^*0)(0 \cup (1\,1^*0))^*$. On ajoute donc une transition de q'_0 à q'_1 avec cette expression. Puis on supprime q_1, ce qui donne l'automate suivant :



Il n'est pas évident, à première vue, que $(1^*0)(0 \cup (1\,1^*0))^*$ décrit le même langage que l'automate de départ, à savoir $(0 \cup 1)^*0$, mais c'est bien le cas ! [Pour les curieux : On peut d'abord réaliser que $(0 \cup (11^*0))^*$ est déjà équivalent à $(0 \cup 1)^*0$: les deux motifs 0 et 11^*0 terminent tous les deux par 0 et le fait de les répéter grâce à l'étoile extérieure capture n'importe quelle façon d'agencer des 1 et des 0 avant le zéro final. Puis l'ajout du préfixe (1^*0) ne rajoute aucune contrainte, le mot peut toujours commencer par 1 ou 0, et si ce préfixe "consomme" le seul 0 d'un mot du langage, alors c'est que c'était le dernier et il suffit de passer zéro fois dans la partie $(0 \cup (11^*0))^*$.]

4.4.2 Conclusion

Au cours de ces dernières séances, nous avons montré que tout AFN peut être transformé en un AFD équivalent (au passage, les AFDs étant des cas particuliers d'AFNs) ; que tout expression régulière (ER) peut être transformée en AFN ; et enfin que tout AFD peut être transformé en ER. Nous avons donc montré, transitivement, que tous ces outils ont la même expressivité : ils décrivent chacun à sa façon des langages réguliers, ni plus ni moins.