Programmation C

Préparation du projet : expressions régulières et automates

Présentation

Le projet de programmation C qui vous sera donné prochainement consistera à implémenter une ***bibliothèque de manipulation d'expressions régulières et d'automates finis*** et des fonctions utilisant cette bibliothèque. Ces objets sont destinés à **traiter du texte**. Par exemple, on implémentera une version de la commande `grep`, qui permet de rechercher un motif dans un tezte. Cette feuille présente les objets qui seront manipulés dans le projet.

De nombreux logiciels sont dédiés à la manipulation de texte. C'est le cas pour les éditeurs (comme Emacs ou VS Code), ou les logiciels permettant de mettre en forme un document (comme Word ou LibreOffice). Une fonctionnalité de base de tels logiciels est de rechercher une chaîne de caractères particulière dans le document. Elle permet de rechercher et d'examiner toutes les occurrences d'un motif recherché. Une autre fonctionnalité courante est de remplacer les occurrences d'un motif de texte par un autre motif.

Il est fréquent qu'une personne qui utilise un tel logiciel soit plus exigeante : plutôt que de rechercher (ou remplacer) une chaîne fixe, on souhaite souvent faire une recherche plus flexible, d'un motif ayant une *forme particulière*. Par exemple, si dans un programme, on a utilisé des identificateurs de la forme `x0`, `x1`, `x2`, etc., on peut vouloir rechercher les chaînes de la forme : « `x` suivi par une suite non vide de chiffres ».

Les expressions régulières nous apportent une syntaxe permettant de spécifier certains motifs. Elles sont très utilisées en informatique, d'un point de vue théorique comme d'un point de vue pratique, même si les logiciels peuvent utiliser des syntaxes différentes. Parmi les commandes *shell* classiques qui utilisent des expressions régulières, on peut citer `grep`, `awk`, `sed`, ou le shell lui-même. Plusieurs langages (par exemple Python, Rust ou Perl), fournissent par ailleurs, nativement ou via des bibliothèques, des outils de manipulation de telles expressions. Enfin, les analyseurs lexicaux, utilisés en particulier dans les compilateurs, sont basés sur les expressions régulières.

4.1 Mots et langages

Dans la suite, on va manipuler des suites de lettres. Chaque lettre appartiendra à un ensemble fini, appelé alphabet. Un ***alphabet*** est donc simplement un ensemble fini de symboles, appelés des ***lettres***. Des exemples d'alphabets sont les suivants :

- $A_1 = \{a, b, c, \ldots, z\}$ est l'alphabet latin de base restreint aux lettres minuscules.
- $A_2 = \{0, 1\}$ est l'alphabet binaire. Il ne contient que deux lettres : 0 et 1. Dans ce contexte, les symboles 0 et 1 sont considérés comme des lettres, et non comme des entiers.
- $A_3 =$ l'alphabet de tous les symboles du code ASCII.

Une fois fixé un alphabet, un ***mot*** est simplement une suite finie de lettres de cet alphabet. Par exemple,

`abracadabra`

est un mot de 11 lettres (certaines étant répétées) sur l'alphabet A_1. Si, dans un langage de programmation, les lettres sont représentées par des caractères, alors les mots correspondent aux *chaînes de caractères*. Par exemple, en C, on noterait la chaîne ci-dessus par `"abracadabra"`. Bien sûr, les mots dans une langue comme le français ont en général une signification. Dans notre cadre par contre, un mot est simplement une suite de lettres, ayant un sens ou non. Par exemple, sur l'alphabet ASCII, la suite de lettres suivantes est un mot de 21 caractères :

`l%am39B)@7nW99*$?>70G`

Il n'a aucun sens mais répond à la définition de mot : c'est juste une suite de caractères. De même, la suite de lettres vide, correspondant à la chaîne notée `""` en C, est un mot valide noté ε.

Mots

Un alphabet A étant fixé, un ***mot*** (sur A) est une suite de lettres de A. On note un mot en juxtaposant simplement chacune de ses lettres, de gauche à droite. Le ***mot vide*** est noté ε. La ***longueur*** d'un mot u se note $|u|$: c'est son nombre total de lettres, c'est-à-dire la longueur de la suite de lettres. Par exemple, le mot $abba$ a pour longueur 4. On note donc $|abba| = 4$. La longueur du mot vide est 0 : on a $|\varepsilon| = 0$.

On manipulera souvent des **ensembles de mots**. Un tel ensemble s'appelle un langage.

Langages

Un ***langage*** sur un alphabet A est un ensemble de mots sur A. L'ensemble de tous les mots sur A se note A^*. Un langage de mots sur A est donc un sous-ensemble de A^*.

Exemple 1

- Le langage vide est celui qui ne contient aucun mot. Il se note $\emptyset$.
- Le langage $\{\varepsilon\}$ ne contient qu'un seul mot : le mot vide.
- Sur l'alphabet $\{a\}$, le langage des mots de longueur paire est l'ensemble (infini) de mots suivant :

$$\{\varepsilon, aa, aaaa, aaaaaa, aaaaaaaa, \ldots\}.$$

- Sur l'alphabet binaire $\{0, 1\}$, le langage des mots qui se terminent par 0 est également infini :

$$\{0, 00, 10, 000, 010, 100, 110, \ldots\}.$$

Soyez rigoureux sur le vocabulaire !

Le mot vide ε n'est pas une lettre ! Ce n'est pas un langage non plus : c'est un mot (le seul mot de longueur 0). Le langage vide, lui, est bien un langage (comme son nom l'indique) : celui qui ne contient aucun mot. Enfin, le langage $\{\varepsilon\}$ est un langage qui contient un unique mot : le mot vide. Ces trois objets sont donc différents.

4.1.1 Opérations sur les langages

Opérations Booléennes

Plusieurs opérations classiques peuvent être faites sur les langages. D'une part, comme les langages sont des ensembles, on peut considérer toutes les opérations Booléennes que l'on peut faire sur les ensembles : union, intersection et complément. Soient K et L deux langages sur un alphabet A, c'est-à-dire que $K \subseteq A^*$ et $L \subseteq A^*$.

- L'***union*** de K et L est le langage,

$$K \cup L = \{w \in A^* \mid w \in K \textbf{ ou } w \in L\}.$$

  Autrement dit, $K \cup L$ est l'ensemble des mots qui appartiennent soit à K, soit à L.
- L'***intersection*** de K et L est le langage,

$$K \cap L = \{w \in A^* \mid w \in K \textbf{ et } w \in L\}.$$

  Autrement dit, $K \cap L$ est l'ensemble des mots qui appartiennent à la fois à K et à L.
- Le ***complément*** de K (dans A^*) est le langage,

$$A^* \setminus K = \{w \in A^* \mid w \notin K\}.$$

  Autrement dit, $A^* \setminus K$ est l'ensemble des mots qui n'appartiennent pas à K.

Autres opérations

On peut exploiter le fait que les langages sont des ensembles **de mots** pour définir d'autres opérations sur les langages. Pour deux mots u et v, on note uv le mot obtenu en « collant » v après u. On dit que le mot uv est la ***concaténation*** de u et de v. Par exemple, si $u = aba$ et $v = bb$, alors uv est le mot $ababb$.

- La ***concaténation*** de K et L est le langage,

$$KL = \{uv \in A^* \mid u \in K \text{ et } v \in L\}.$$

  Autrement dit, KL est l'ensemble des mots de la forme uv avec u dans K et v dans L. C'est donc l'ensemble de tous les mots obtenus en concaténant un mot de K avec (à sa suite) un mot de L.
  La concaténation de deux langages étant maintenant définie, on peut définir la puissance n-ième d'un langage : elle est définie par récurrence sur n de la façon suivante :
  - $L^0 = \{\varepsilon\}$,
  - pour $n \geqslant 1$, $L^n = LL^{n-1}$.

  On peut vérifier que L^n est l'ensemble des mots de la forme $w_1 w_2 \cdots w_n$ où chaque w_i est dans L.
- L'***étoile de Kleene*** de K (dans A^*) est le langage,

$$K^* = \bigcup_{n \in \mathbb{N}} K^n = K^0 \cup K^1 \cup K^2 \cup K^3 \cup K^4 \cup \cdots$$

  Autrement dit, un mot est dans K^* si c'est une concaténation d'un nombre quelconque de mots de K.

Exercice 1

On se place sur l'alphabet $A = \{a, b\}$.

1. Écrire les mots du langage $\{a, bb\}\{a, bab, bb\}$, et les 7 mots les plus courts du langage $L = (\{a\} \cup \{bb\})^*$.
2. Décrire en français le langage L.
3. Écrire une expression utilisant les opérations données ci-dessus, **sauf le complément**, décrivant le langage $A^* \setminus L$.

4.2 Expressions régulières

Une expression régulière représente un langage. Les expressions régulières sont formées en utilisant les lettres et les symboles « $\emptyset$ », « $+$ », « $*$ » et « $\cdot$ ». La notation est choisie pour qu'elle soit intuitive. Par exemple, « $\emptyset$ » dans une expression représente l'ensemble vide $\emptyset$ et « $*$ » représente l'étoile de Kleene.

4.2.1 Syntaxe des expressions régulières

Syntaxe des expressions régulières

- Il y a deux sortes d'expressions de base :
  - $\emptyset$ est une expression régulière.
  - a est une expression régulière pour chaque lettre $a \in A$.
- Les autres règles, récursives, permettent de former de nouvelles des expressions régulières :
  - si E_1 et E_2 sont des expressions régulières, alors $(E_1 + E_2)$ est une expression régulière.
  - si E_1 et E_2 sont des expressions régulières, alors $(E_1 \cdot E_2)$ est une expression régulière.
  - si E est une expression régulière, alors (E^*) est une expression régulière.

On peut alléger la notation en supprimant certaines parenthèses. Pour cela, on affecte une priorité aux symboles « $+$ », « $*$ » et « $\cdot$ » : le symbole « $*$ » est plus prioritaire que « $\cdot$ », lui-même plus prioritaire que « $+$ ». Ainsi par exemple, $a + (b \cdot (c^*))$ s'écrit plus simplement $a + b \cdot c^*$. De plus, plutôt que d'écrire $E_1 \cdot E_2$, on écrit $E_1 E_2$. L'expression ci-dessus s'écrit donc aussi $a + bc^*$.

Exemple 2 - Exemples d'expressions régulières

Les expressions suivantes sont des expressions régulières bien formées.

- $\emptyset$,
- $\emptyset^*$,
- $(aa)^*$,
- $\emptyset(aa)^*$,
- $(0+1)^*0$.

À ce stade, on a simplement défini ce que sont les expressions régulières bien formées (*i.e.*, la ***syntaxe*** des expressions). On doit maintenant définir pour une expression ce qu'elle signifie (*i.e.*, sa ***sémantique***). Une expression représente un **langage**. La définition est récursive (et attendue).

Sémantique des expressions régulières

On associe à chaque expression régulière E un langage $\mathcal{L}(E)$ de la façon suivante :

- Expressions de base :
  - $\emptyset$ représente le langage vide $\emptyset$: $\mathcal{L}(\emptyset) = \emptyset$.
  - a représente le langage $\{a\}$: $\mathcal{L}(a) = \{a\}$.
- Les autres règles permettent donner un sens aux expressions régulières formées récursivement :
  - si E_1 est une expression régulière qui représente le langage L_1 et si E_2 est une expression régulière qui représente le langage L_2, alors $(E_1 + E_2)$ représente $L_1 \cup L_2$.

$$\mathcal{L}(E_1 + E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2).$$

  - si E_1 est une expression régulière qui représente le langage L_1 et si E_2 est une expression régulière qui représente le langage L_2, alors $(E_1 \cdot E_2)$ représente $L_1 L_2$.

$$\mathcal{L}(E_1 \cdot E_2) = \mathcal{L}(E_1)\mathcal{L}(E_2).$$

  - si E est une expression régulière qui représente le langage L, alors (E^*) représente L^*.

$$\mathcal{L}(E^*) = \mathcal{L}(E)^*.$$

Il est courant d'alléger encore une fois la notation, et de parler « du langage ab^* », alors qu'on devrait dire, en toute rigueur, « le langage $\mathcal{L}(ab^*)$ ». Autrement dit, le contexte nous permet de savoir si on parle d'une expression ou du langage représenté par une expression.

Exercice 2

Exemples de langages descriptibles par des expressions

1. Pour chacune des expressions régulières de l'exemple 2, exprimer en français le langage représenté par cette expression.
2. Pour chacun des langages suivants sur l'alphabet $\{a, b\}$, donner une expression régulière qui le représente :
   (a) qui contiennent un nombre pair de a.
   (b) qui ne commencent pas par ab.
   (c) qui contiennent le bloc ab,
   (d) qui ne contiennent pas le bloc ab.

Exercice 3

Expressions réguilères sans étoile de Kleene

Soit L un langage représenté par une expression régulière qui n'utilise pas l'étoile de Kleene.

1. Quelle propriété particulière le langage L possède-t-il ?
2. Inversement, justifier que tout langage qui a la propriété énoncée à la question 1 peut être représenté par une expression régulière qui n'utilise pas l'étoile de Kleene.

4.3 Automates finis

Les automates finis sont un modèle de calcul : chaque automate représente un programme. On peut ainsi coder un automate en C, Python, OCaml ou tout autre langage de programmation. Par rapport aux programmes réalistes, les possibilités des automates sont cependant très limitées. En particulier, un automate ne fait que lire une suite de lettres (donc un mot), et, une fois ce mot lu, émet un verdict oui/non : le mot est accepté ou rejeté. Comme les expressions régulières, un automate fini définit donc un langage (celui des mots qu'il accepte), que l'on appelle ***langage reconnu*** par l'automate.

Syntaxe. Un automate fini est un quintuplet (A, Q, I, F, δ) où :

- A est l'alphabet d'entrée, il contient les symboles avec lesquels on écrit l'entrée de l'automate.
- Q est un ensemble fini d'états.
- $I \subseteq Q$ est l'ensemble des états initiaux (ceux dans lesquels le calcul peut commencer).
- $F \subseteq Q$ est l'ensemble des états finaux ou acceptants (ceux dans lesquels un calcul acceptant peut s'arrêter).
- δ est la relation de transition. C'est elle qui constitue le « programme » de l'automate. On a :

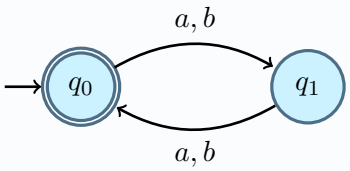
$$\delta \subseteq Q \times A \times Q.$$

?

Plutôt que d'écrire que $(p, a, q) \in \delta$, on écrit souvent $p \xrightarrow{a} q$. On écrit aussi $p \xrightarrow{a,b} q$ pour indiquer qu'il y a deux transitions $p \xrightarrow{a} q$ et $p \xrightarrow{b} q$. De la même façon, on représente graphiquement un automate fini, par un graphe orienté : les sommets sont les états, et on indique les transitions entre états par des arcs étiquetés par la lettre impliquée dans la transition. On indique les états initiaux par une flèche entrante, et les états finaux par une flèche sortante.

Exercice 4

1. Donner la liste des transitions de l'automate suivant.



2. Dessiner l'automate $\mathcal{A} = (A, Q, I, F, \delta)$ sur l'alphabet $A = \{a, b\}$, avec $Q = \{q_0, q_1, q_2, q_3\}$, $I = \{q_0\}$, $F = \{q_3\}$ et $\delta = \Big\{(q_0, a, q_1), (q_1, b, q_2), (q_2, a, q_2), (q_2, b, q_2), (q_2, b, q_3)\Big\}$.

Sémantique. Considérons un automate $\mathcal{A} = (A, Q, I, F, \delta)$. Pour tout mot $w \in A^*$, on va définir ce qu'est un calcul de $\mathcal{A}$ sur le mot d'entrée w (le mot *sémantique* fait référence à tout ce qui concerne le calcul d'une machine, par opposition à sa *syntaxe* qui fait référence au texte du programme).

Soit $w = a_1 \cdots a_n \in A^*$ avec $a_1, \ldots, a_n \in A$. Le mot w est donc de longueur n. Un ***calcul partiel*** sur w dans l'automate $\mathcal{A}$ est une suite de $n+1$ états $q_0, \ldots, q_n \in Q$ telle que pour tout $i \geqslant 1$, on a $(q_{i-1}, a_i, q_i) \in \delta$. Avec la notation précédente, on représente un tel calcul partiel de la façon suivante :

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} q_n.$$

On dit que $w = a_1 \cdots a_n$ est ***l'étiquette*** du calcul partiel.

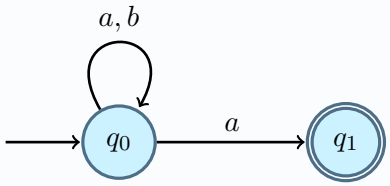
On dit que q_0 est le ***premier état*** de ce calcul partiel et que q_n est son ***dernier état***. Si $q, r \in Q$ sont des états et v est un mot de A^*, on note $q \xrightarrow{v} r$ s'il existe un calcul partiel sur v dans $\mathcal{A}$ allant de q à r, c'est-à-dire dont le premier état est q et le dernier état est r.

Un ***calcul*** est un calcul partiel dont le premier état est initial, c'est-à-dire dans I. Un calcul est ***acceptant*** si son dernier état est final, c'est-à-dire dans F.

Exercice 5

Calculs dans un automate

On considère l'automate suivant.



1. Donner un calcul sur le mot aba.
2. Le calcul donné à la question précédente est-il unique ?
3. Décrire en français l'ensemble des mots qui étiquettent un calcul acceptant dans cet automate.

Langage accepté (ou reconnu) par un automate

On dit qu'un ***mot est accepté***, ou ***reconnu*** par l'automate $\mathcal{A}$ **s'il existe** un calcul acceptant sur ce mot dans $\mathcal{A}$. Notez qu'il peut aussi y avoir d'autres calculs sur ce même mot qui ne sont pas acceptants. Le ***langage accepté***, ou ***reconnu*** par l'automate $\mathcal{A}$ est l'ensemble $\mathcal{L}(\mathcal{A})$ des mots acceptés par $\mathcal{A}$. Notez qu'un même langage peut être reconnu par plusieurs automates distincts. Enfin, on dira qu'un langage est ***régulier*** s'il existe un automate qui le reconnaît.

Exercice 6

Construction d'automates pour des langages particuliers

1. Quel est le langage reconnu par l'automate de l'exercice 4 ? En donner une expression régulière.
2. Pour chacun des langages suivants, construire un automate qui le reconnaît.
   a. Le langage des mots sur l'alphabet $\{a, b, c\}$ qui se terminent par $abba$.
   b. Le langage des mots sur l'alphabet $\{a, b, c\}$ qui contiennent $abba$.
   c. Le langage des mots sur l'alphabet $\{a, b, c\}$ de longueur impaire.
   d. Le langage des mots sur l'alphabet $\{a, b, c\}$ qui ont un nombre pair de c.
   e. Le langage des mots sur l'alphabet $\{a, b\}$ qui contiennent aba.
   f. Le langage des mots sur l'alphabet $\{a, b\}$ qui ne contiennent pas aba.
   g. Le langage des mots sur l'alphabet $\{0, 1, 2\}$ qui représentent un entier divisible par 2 en base 3.

4.4 Des expressions régulières aux automates finis

À ce stade, nous avons deux formalismes différents pour représenter un langage :

- les expressions régulières,
- les automates finis.

L'avantage des expressions régulières est qu'elles permettent de spécifier assez facilement des langages. Par contre, elles ne sont pas faciles à manipuler algorithmiquement. Inversement, l'avantage des automates est qu'il y a plusieurs algorithmes pour les manipuler (nous en verrons quelques uns plus loin), mais il peut être compliqué de trouver un automate pour spécifier un langage qui nous intéresse.

Dans cette partie, nous allons voir que toute expression régulière peut être « compilée » en un automate qui reconnaît le langage représenté par l'expression. Nous ne prouverons pas la correction de cet algorithme. En revanche, il est important de le comprendre, car on demandera de l'implémenter dans le projet C. On va d'abord illustrer cet algorithme, dû à Glushkov, sur un exemple.

Avant de commencer, une définition : on dit qu'un mot u est un ***facteur*** d'un mot w s'il existe deux mots (éventuellement vides) x, y tels que $w = xuy$. Par exemple, les facteurs de abc sont $\varepsilon, a, b, c, ab, bc, abc$.

Nous sommes prêts pour expliquer l'algorithme de Glushkov en l'illustrant sur un exemple particulier : l'expression $E_{\text{orig}} = a(ab + b)^*a^*$. Notre objectif est de construire un automate reconnaissant ce langage.

Étape 1. On renomme les lettres pour qu'après renommage, chaque lettre apparaisse une **unique** fois. Ainsi, l'expression $E_{\text{orig}} = a(ab+b)^*a^*$ peut être renommée en $E_{\text{renamed}} = a_1(a_2b_1 + b_2)^*a_3^*$.

Étape 2. On calcule plusieurs informations pour chaque sous-expression E de l'expression ainsi renommée :

(a) On détermine si le mot vide ε est dans le langage $\mathcal{L}(E)$ représenté par E. Sur notre exemple, le mot vide appartient au langage de trois sous-expressions :

- $(a_2b_1 + b_2)^*$,
- a_3^*,
- $(a_2b_1 + b_2)^*a_3^*$.

Il n'appartient à aucun des langages des autres sous-expressions, comme par exemple a_1, a_2b_1 ou $a_1(a_2b_1 + b_2)^*$, ou encore l'expression complète $a_1(a_2b_1 + b_2)^*a_3^*$.

(b) On calcule l'ensemble $\mathsf{First}(E)$ des lettres (renommées) telles qu'il existe un mot de $\mathcal{L}(E)$ commençant par cette lettre. Par exemple, $\mathsf{First}((a_2b_1 + b_2)^*) = \{a_2, b_2\}$, car tout mot du langage $(a_2b_1 + b_2)^*$ commence soit par a_2, soit par b_2.

(c) On calcule symétriquement l'ensemble $\mathsf{Last}(E)$ des lettres (renommées) telles qu'il existe un mot de $\mathcal{L}(E)$ finissant par cette lettre. Par exemple, $\mathsf{Last}((a_2b_1 + b_2)^*) = \{b_1, b_2\}$, car tout mot du langage $(a_2b_1 + b_2)^*$ se termine soit par b_1, soit par b_2.

(d) On calcule enfin l'ensemble $\mathsf{Follow}(E)$ des mots de longueur 2 pouvant apparaître comme facteur d'un mot de $\mathcal{L}(E)$. Par exemple, $\mathsf{Follow}((a_2b_1 + b_2)^*) = \{a_2b_1, b_1a_2, b_1b_2, b_2a_2, b_2b_2\}$.

Un calcul récursif

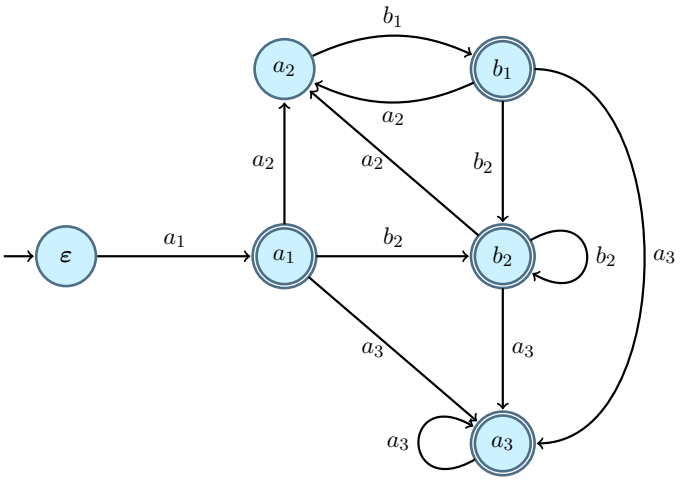
Les calculs sont effectués pour chaque sous-expression, mais l'objectif est de calculer ces informations uniquement pour l'expression de départ renommée E_{renamed}. On peut se demander alors pourquoi le faire pour chacune des sous-expressions de E_{renamed}. La raison est que le calcul est récursif. Par exemple, si E_{renamed} est une concaténation de deux sous-expressions E_1 et E_2, l'algorithme de calcul s'appellera récursivement sur E_1 et E_2.

Une fois ce calcul fait, on construit un automate reconnaissant E_{renamed} de la façon suivante :

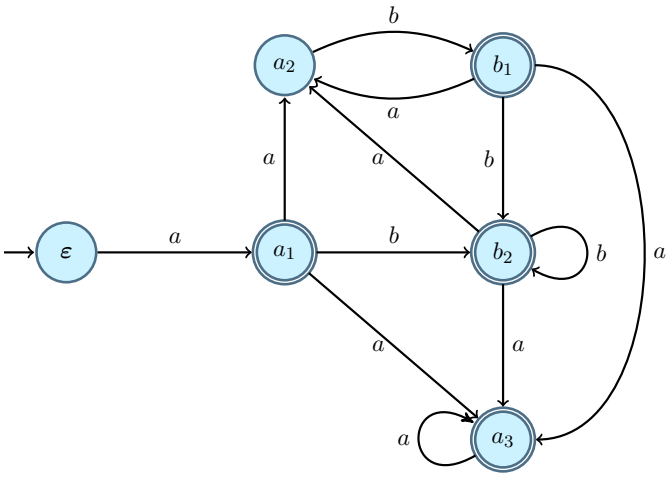
- Ses **états** sont étiquetés par les lettres apparaissant dans l'expression E_{renamed}, et par le mot vide. Dans notre cas, il y a 5 lettres, il y aura donc 6 états, étiquetés $\{\varepsilon, a_1, a_2, a_3, b_1, b_2\}$. Intuitivement, chaque état mémorise la dernière lettre lue (ou ε si le calcul n'a pas encore commencé).
- L'**unique état initial** est ε.
- Les **états finaux** sont ceux qui sont dans $\mathsf{Last}(E_{\text{renamed}})$, auxquels on ajoute l'état ε si $\varepsilon \in \mathcal{L}(E_{\text{renamed}})$. Dans notre cas, on peut vérifier que $\mathsf{Last}(a_1(a_2b_1 + b_2)^*a_3^*) = \{a_1, b_1, b_2, a_3\}$ et $\varepsilon \notin \mathcal{L}(E_{\text{renamed}})$.
- Les **transitions** relient :
  - ε à tous les états qui sont dans $\mathsf{First}(E_{\text{renamed}})$,
  - un état x à un état y (où x et y sont deux lettres renommées) si le mot de deux lettres xy appartient à $\mathsf{Follow}(E_{\text{renamed}})$.

  Enfin, pour toute lettre ℓ, une transition entrant dans l'état ℓ est toujours étiquetée ℓ.

Dans notre exemple, on obtient l'automate suivant :



Il ne reste plus qu'à revenir aux lettres originales de l'expression E_{orig}, sur les transitions pour obtenir l'automate suivant :



Information

L'automate obtenu à partir d'une expression E a $|E|+1$ états (où $|E|$ est le nombre de lettres de l'expression E). En revanche, il est, par construction, non déterministe, parce qu'on a supprimé les indices des lettres. Il peut donc y avoir plusieurs transitions partant d'un même état et étiquetées avec la même lettre.

Exercice 7

Calcul des ensembles First, Last et Follow

On se donne une expression sur un alphabet A. Proposez des algorithmes pour calculer les informations nécessaires pour l'algorithme de Glushkov. Autrement dit, pour toute sous-expression E de l'expression renommée, proposez un algorithme pour :

1. Déterminer si $\varepsilon \in \mathcal{L}(E)$,
2. Calculer l'ensemble $\mathsf{First}(E)$, c'est-à-dire déterminer, pour chaque $a \in A$ s'il existe un mot commençant par a dans $\mathcal{L}(E)$.
3. Calculer l'ensemble $\mathsf{Last}(E)$, c'est-à-dire déterminer, pour chaque $a \in A$, s'il existe un mot se terminant par a dans $\mathcal{L}(E)$.
4. Calculer l'ensemble $\mathsf{Follow}(E)$, c'est-à-dire déterminer, pour chaque couple $(a,b) \in A \times A$ s'il existe un mot contenant le facteur ab dans $\mathcal{L}(E)$.

Stratégie pour concevoir ces algorithmes. Comme indiqué plus haut, choisir d'exprimer chacun des 4 algorithmes de façon récursive est bien adapté ici. En effet, les expressions régulières sont elles-mêmes définies récursivement. Vous devrez donc considérer les cas de base (E est l'expression $\emptyset$ ou E est une lettre), puis les cas « inductifs » (E est de la forme $E_1 + E_2$, E_1E_2 ou E_1^*).

Exercice 8

Application de l'algorithme de Glushkov

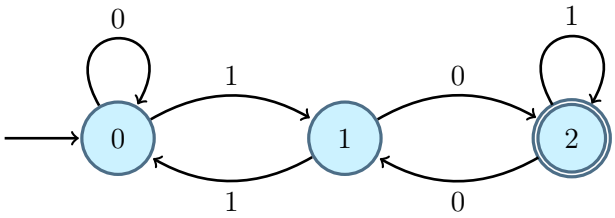
Appliquer l'algorithme de Glushkov aux expressions régulières suivantes :

1. $aa(a+bb)^*b$,
2. $a((ab)^*ca^*b^*)^*$.

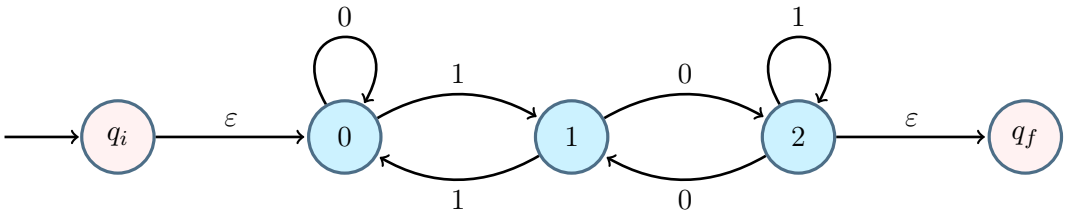
4.5 Des automates finis aux expressions régulières

Dans cette partie, nous allons voir un algorithme très simple qui prend en entrée un automate fini $\mathcal{A}$, et qui produit une expression régulière représentant le langage $\mathcal{L}(\mathcal{A})$. Nous ne prouverons pas l'algorithme, mais cela est facile. À nouveau, l'algorithme sera illustré sur un exemple. Dans cet algorithme, on se permet

d'étiqueter les transitions des automates pas seulement par des lettres, mais plus généralement par des expressions régulières. On illustre la construction sur l'automate suivant.

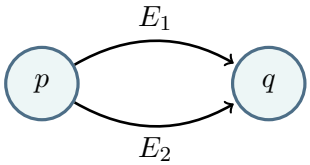


La première étape consiste à ajouter un nouvel état q_i qui devient l'unique état initial de l'automate, et on ajoute des transitions étiquetées par ε entre q_i et tout état qui était initial dans l'automate d'origine. Symétriquement, on ajoute un nouvel état q_f qui devient le seul état final, et des transitions étiquetées par ε entre tout état qui était final dans l'automate d'origine et q_f. Sur l'exemple, on obtient :

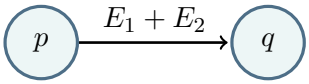


Ensuite, on effectue répétitivement une des transformations suivantes, qui font décroître le nombre de transitions ou le nombre d'états, jusqu'à ce qu'il ne reste plus que deux états : q_i et q_f reliés par une unique transition. L'algorithme garantit que l'expression qui étiquette cette dernière transition restante est une expression régulière qui représente le langage reconnu par l'automate d'origine. L'ordre dans lequel on applique ces transitions n'a pas d'importance concernant la correction de cet algorithme.

La **première transformation** est la suivante. Elle fait diminuer le nombre de transitions.

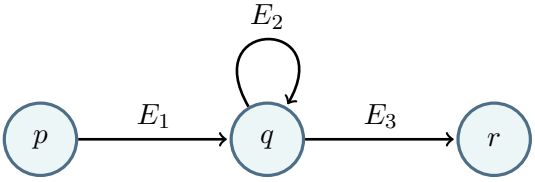


Avant transformation

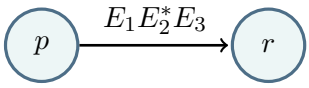


Après transformation

La **seconde transformation** illustrée ci-dessous fait diminuer les nombres de transitions et d'états. Dans cette transformation, on élimine un état q tel qu'il y a une unique transition $q \xrightarrow{E_2} q$. Si une telle transition n'existe pas, on peut en ajouter une avec $E_2 = \varepsilon$. Inversement, s'il y en a plusieurs, on peut les fusionner au préalable en utilisant la première transformation. L'application de la seconde transformation crée une transition **pour chaque couple** (p, r) quand il existe une transition de p à q et une transition de q à r.

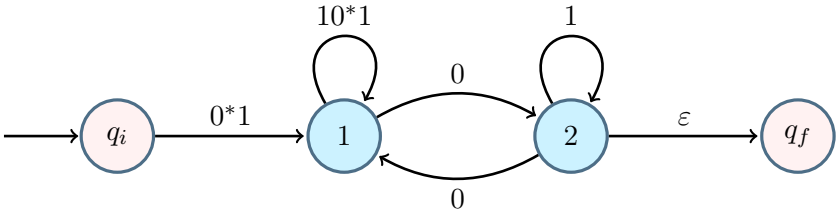


Avant transformation

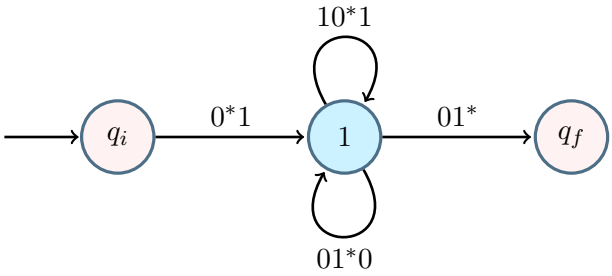


Après transformation

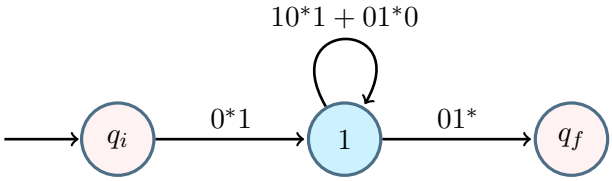
Par exemple, on peut appliquer la seconde transformation pour éliminer l'état 0 de l'automate donné ci-dessus. Le rôle du couple (p, r) peut être joué par le couple $(q_i, 1)$ d'une part, et par le couple $(1, 1)$ d'autre part. On obtient l'automate suivant :



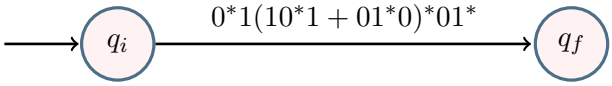
On peut maintenant éliminer l'état 2, par exemple. On obtient :



On peut maintenant appliquer la première transformation entre l'état 1 et lui-même :



Enfin, on élimine l'état 1, ce qui donne :



Le langage reconnu par l'automate de départ est donc aussi représenté par l'expression régulière

$$0^*1(10^*1 + 01^*0)^*01^*.$$

Bien sûr, il n'y a pas qu'une seule expression représentant un langage. L'algorithme ci-dessus, dû à Brzozowski et McCluskey, produit une expression qui dépend des choix des transformations effectuées.

Récap : Automates finis et expressions régulières décrivent les mêmes langages

Les Sections 4.4 et 4.5 montrent que les expressions régulières et les automates finis décrivent exactement les mêmes langages. En effet, on a vu que :

- On peut passer d'une expression régulière à un automate fini qui reconnaît le langage représenté par l'expression (par l'algorithme de Glushkov).
- Inversement, on peut passer d'un automate fini à une expression qui représente le langage reconnu par l'automate (par l'algorithme de Brzozowski et McCluskey).

Ceci explique la terminologie introduite plus haut : les langages décrits par les expressions régulières ou les automates finis s'appellent ***langages réguliers***.

4.6 Algorithmes sur les automates

Dans cette partie, on s'intéresse à des problèmes sur les automates. On commence par deux questions naturelles. La première consiste à déterminer si un mot est accepté par un automate, ce qui est lié à notre motivation : cela nous permet déjà de rechercher un motif dans un texte.

Exercice 9

Proposez des algorithmes (informellement, en français) pour répondre aux deux problèmes ci-dessous.

Problème 1 Étant donnés un mot et un automate, déterminer si le mot est accepté par l'automate.

Problème 2 Étant donné un automate, déterminer si le langage accepté par l'automate est vide ou non.

On va maintenant s'intéresser aux propriétés de clôture des automates de façon algorithmique. On veut montrer que si on dispose de deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$, on peut construire des automates reconnaissant l'union et l'intersection des langages $\mathcal{L}(\mathcal{A}_1)$ et $\mathcal{L}(\mathcal{A}_2)$.

Exercice 10 — Automates reconnaissant l'union et l'intersection de langages reconnus par des automates

On se donne deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$.

1. Proposer un algorithme (très simple !) construisant un automate reconnaissant $\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$ à partir des automates $\mathcal{A}_1$ et $\mathcal{A}_2$.
2. Proposer un algorithme construisant un automate reconnaissant $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ à partir des automates $\mathcal{A}_1$ et $\mathcal{A}_2$. Pour cette construction, vous pouvez considérer des états qui mémorisent à la fois un état de $\mathcal{A}_1$ et un état de $\mathcal{A}_2$.

L'exercice précédent montre que l'union et l'intersection de deux langages reconnus par automates sont aussi reconnues par automates. Il est naturel de se demander si la même propriété est valable pour le complément. Autrement dit, on voudrait savoir si, étant donné un automate $\mathcal{A}$ sur un alphabet A, on peut construire un nouvel automate qui reconnaît le langage $A^* \setminus \mathcal{L}(\mathcal{A})$.

Exercice 11 — Le dilemme du complément

L'informaticien Alan T. souhaite construire, à partir d'un automate $\mathcal{A}$ sur un alphabet A, un nouvel automate qui reconnaît le langage $A^* \setminus \mathcal{L}(\mathcal{A})$. Comme il constate qu'un mot accepté possède au moins un calcul qui se termine dans un état final, il a l'idée d'inverser le statut des états : si $\mathcal{A} = (A, Q, I, F, \delta)$, il produit l'automate $\mathcal{B} = (A, Q, I, Q \setminus F, \delta)$. Ainsi, les deux automates ne diffèrent que par leurs états finaux (ceux de $\mathcal{B}$ sont exactement ceux qui ne sont pas finaux dans $\mathcal{A}$).

Montrer que la construction n'est pas correcte, en fournissant deux exemples.

1. Dans le premier exemple, on demande que l'union $\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$ ne soit pas égale à A^*.
2. Dans le second exemple, on demande que l'intersection $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$ ne soit pas vide.

Une première raison qui fait échouer l'approche précédente est qu'un mot peut n'avoir aucun calcul (acceptant ou non) dans l'automate. Si on applique la construction précédente, le mot ne pourra être lu ni dans $\mathcal{A}$, ni dans l'automate $\mathcal{B}$ calculé en inversant le statut (final ou non) des états. Autrement dit, il peut manquer des transitions pour pouvoir lire le mot.

On dit qu'un automate est *complet* si pour tout état p et toute lettre a, il existe **au moins** une transition étiquetée a issue de p, c'est-à-dire de la forme $p \xrightarrow{a} q$.

Exercice 12 — Complétion d'un automate

Proposer un algorithme qui,

- prend en entrée un automate fini quelconque $\mathcal{A}$, et
- renvoie un automate **complet** $\mathcal{B}$ qui reconnaît le même langage que $\mathcal{A}$.

Une seconde raison qui fait échouer l'approche précédente est qu'un mot peut être accepté en ayant

au moins un calcul acceptant, et au moins un calcul non acceptant. Dans ce cas, il sera aussi accepté par l'automate $\mathcal{B}$, qui ne peut donc pas accepter le complément du langage $\mathcal{L}(\mathcal{A})$.

Automates déterministes

On dit qu'un automate (A, Q, I, F, δ) est ***déterministe*** lorsque les deux conditions suivantes sont satisfaites :

- l'ensemble I ne contient qu'un seul état,
- pour tout état $p \in Q$ et toute lettre $a \in A$, il existe **au plus** une transition étiquetée a issue de p, c'est-à-dire de la forme $p \xrightarrow{a} q$.

Exercice 13 — Automate reconnaissant le complément d'un langage reconnu par un déterministe complet

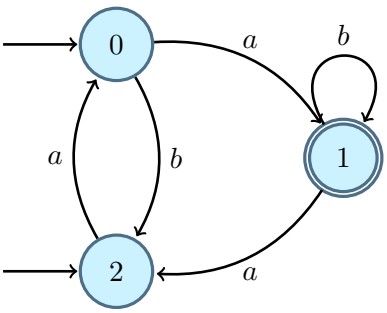
Soit $\mathcal{A} = (A, Q, I, F, \delta)$ un automate complet *et* déterministe.

1. Soit $w \in A^*$ un mot arbitraire. Que peut-on dire du nombre de calculs de $\mathcal{A}$ sur w ?
2. On suppose que L est reconnu par un automate *complet et déterministe*. Décrire un automate qui reconnaît $A^* \setminus L$.

4.6.1 Déterminisation

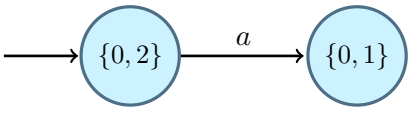
On va maintenant décrire un algorithme que vous aurez également à implémenter dans le projet C : la déterminisation. Cet algorithme prend en entrée un automate et renvoie un automate qui reconnaît le même langage, et qui, de plus, est **déterministe**. En utilisant les résultats des sections précédentes, cet algorithme fournit un résultat intéressant et qui n'est pas évident sur la définition des expressions : le complémentaire du langage représenté par une expression régulière peut lui-même être représenté par une expression régulière.

À nouveau, on explique l'idée de l'algorithme et l'algorithme lui-même sur un exemple. Considérons l'automate fini $\mathcal{A}$ suivant sur l'alphabet $A = \{a, b\}$:

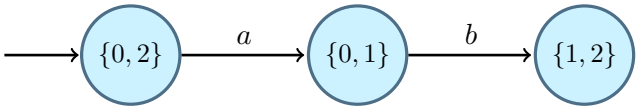


Cet automate n'est pas déterministe car il a deux états initiaux. On décrit la construction pour construire un automate complet et déterministe $\mathcal{B}$ reconnaissant le même langage que $\mathcal{A}$. Imaginons qu'on souhaite savoir si le mot *abab* est accepté par l'automate. Par définition, cela est vrai s'il existe une façon de lire le mot dans l'automate pour atteindre un état final. Pour le détecter, on peut maintenir l'**ensemble des états** que l'on peut atteindre après lecture de chacune des lettres du mot. Au départ, on est soit dans l'état 0, soit dans l'état 2. On mémorise cette information dans le sous-ensemble d'états $\{0, 2\}$.

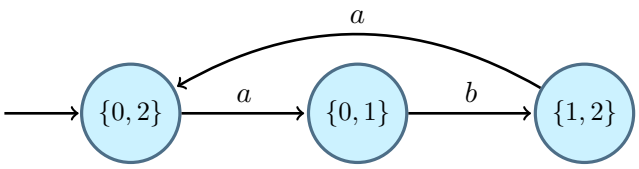
En lisant ensuite le premier « a », on peut atteindre soit l'état 0, soit l'état 1. On mémorise donc cela dans le sous-ensemble $\{0, 1\}$. On peut même commencer à concevoir un début de construction de l'automate qui nous intéresse :



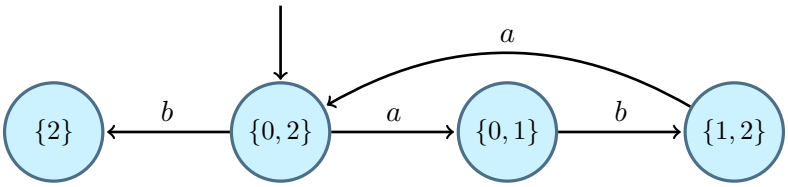
Cet automate mémorise que le calcul commence dans l'un des états 0 ou 2 de l'automate original, et qu'après avoir lu « a » à partir de l'un de ces états, on est soit en 0, soit en 1 dans l'automate original. En continuant cette construction, on obtient, après lecture de la deuxième lettre de *abab* :



Si on ajoute maintenant la troisième lettre (qui est un « a »), on constate qu'on revient sur un état déjà obtenu : l'état initial. On a donc la situation suivante :

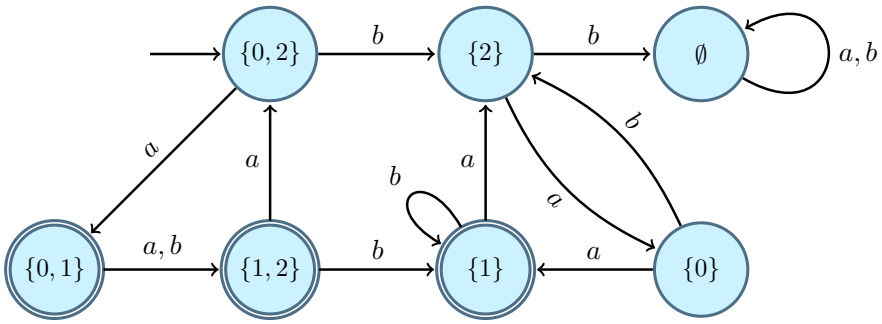


Enfin, si depuis l'état courant $\{0, 2\}$, on lit la dernière lettre de *abab*, on ne peut aller que dans l'état $\{2\}$:



Comme aucun des états du sous-ensemble $\{2\}$ n'est final dans l'automate original, le mot *abab* n'est pas accepté. Dans le nouvel automate, on déclare final un sous-ensemble qui **contient** un état final de l'automate original.

La construction de l'automate déterministe suit ce principe, sans fixer un mot d'entrée. Pour chaque état obtenu et chaque lettre, on calcule l'ensemble des états accessibles en lisant cette lettre dans l'automate original à partir d'un des états du sous-ensemble dont on part. Si on continue l'exemple jusqu'au bout, cela produira l'automate déterministe suivant :



Formellement, soit $\mathcal{A} = (A, Q, I, F, \delta)$ un automate fini. On construit un automate déterministe $\mathcal{B} = (A, \mathcal{Q}, \mathcal{I}, \mathcal{F}, \Delta)$ équivalent à $\mathcal{A}$ de la façon suivante :

- Les états de B sont les sous-ensembles de Q. Autrement dit, $\mathcal{Q} = 2^Q$.
- L'unique état initial $\mathcal{I}$ est le sous-ensemble I. Autrement dit, $\mathcal{I} = \{I\}$.
- Un état de $\mathcal{B}$ est final s'il contient un état final de $\mathcal{A}$. Autrement dit, $\mathcal{F} = \{P \subseteq Q \mid P \cap F \neq \emptyset\}$.
- Pour P, R deux états de $\mathcal{B}$, on a une transition $P \xrightarrow{a} R$ dans Δ si et seulement si

$$R = \{r \in Q \mid \exists p \in P \text{ avec } p \xrightarrow{a} r\}.$$

Construction seulement des états accessibles

Dans l'exemple précédent, il manque un état : $\{0, 1, 2\}$. Cependant, cet état n'est pas accessible depuis l'état initial. On peut donc le supprimer dans la construction, pour obtenir l'automate présenté ci-dessus.

Exercice 14

Déterminisation

Construire des automates déterministes sur l'alphabet $\{a, b\}$ reconnaissant les langages suivants :

- L'ensemble des mots qui ne contiennent pas le facteur $ababa$.
- L'ensemble des mots dont l'avant-avant-dernière lettre est un a.