**Correction des exercices de TD**

# Table des matières



# 1 Analyse Lexicale

## 1.1 TD/TP 1

**Exercice 1 (théorie des langages)**

1. Comment peut-on caractériser un langage rationnel (régulier) ?
2. Les langages de programmation (C, Java, Python) sont-ils réguliers ? Pourquoi ?
3. Comment peut-on caractériser un langage algébrique ?
4. Soit l'extrait suivant de grammaire EBNF de requête de consultation SQL :

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
    select_expr [, select_expr] ...
    [into_option]
    [FROM table_references
        [PARTITION partition_list]]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
        [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
        [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    ;
```

   Expliquer les différentes conventions de notation utilisées !
5. Les requêtes suivantes sont-elles correctes ?

```
select 5 ;
SELECT distinct nom from etudiant
select ref from articles order by rayon asc, ref asc;
```

1. Tout langage reconnu par un automate d'état fini déterministe est un langage rationnel (régulier).
2. Les langages de programmation ne sont pas des langages réguliers car ils contiennent une structure emboîtée non bornée (aucun automate d'état fini déterministe ne peut être construit pour le langage $a^n b^n$).
3. Langages a association de symboles (structure emboîtée non bornée), reconnus par un automate à pile avec un arbre de dérivation constructible par induction.
4. EBNF = Extended Backus-Naur Form. Les `[]` sont utilisés pour désigner une option facultative, | pour le « ou », $\cdots$ pour l'opération de répétition de $n$ caractères ($n \geq 0$) et `{}` pour indiquer un choix non optionnel.
5. Seule la seconde requête n'est pas correcte (il manque le `;`).

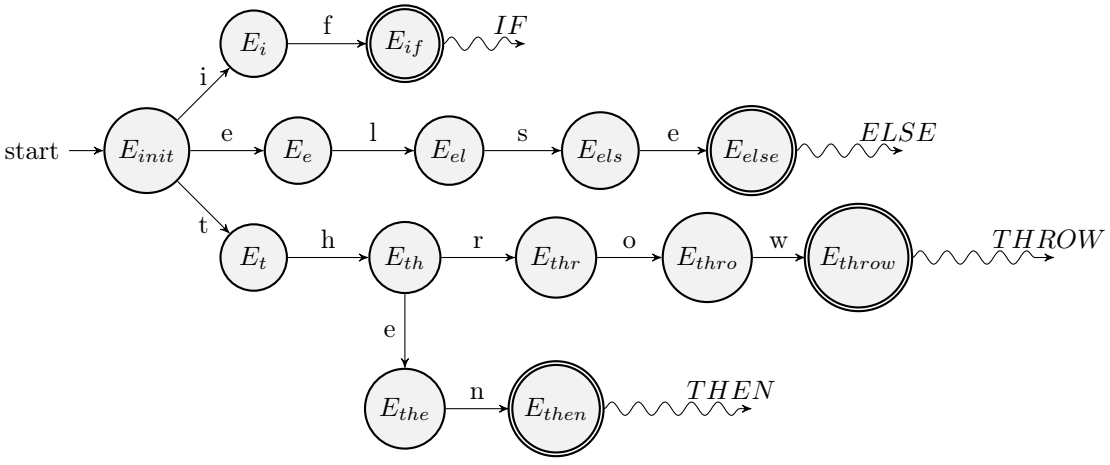
**Exercice 2 (automate)**

— Dessiner un Automate à états Finis Déterministe (AFD) distinct pour chacun des langages suivants :

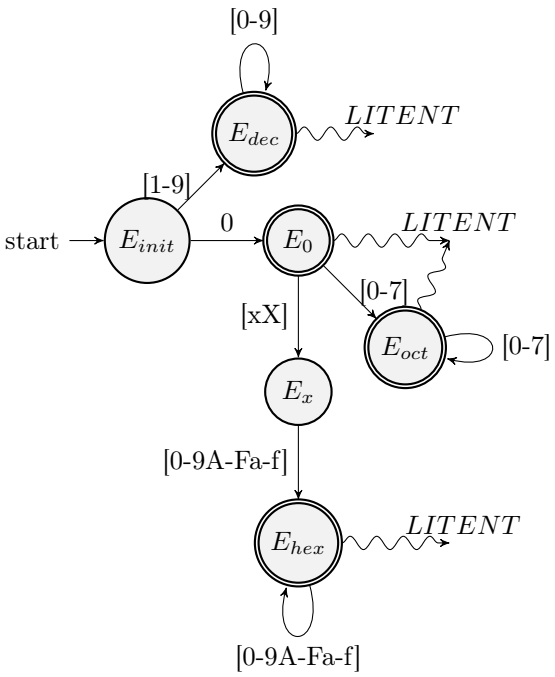
1. Langage de dertains mots-clés du C : $L_{key} = \{if, then, else, throw\}$ (sensible à la casse).
2. Langage des littéraux numériques entiers du C (ou C++, ou Java), décimaux $L_{c10}$ , octaux $L_{c8}$, hexadécimaux $L_{c16}$.
3. Langage $L_{id}$ des identificateurs composés d'une lettre au moins, éventuellement suivie de chiffres, de lettres et de "_".
4. Langage des littéraux numériques flottants décimaux $L_f$ ; la suite de chiffres à gauche ou bien à droite du point décimal pouvant être vide ; l'exposant entier n'est pas obligatoire. Exemples : 13., 1.7e23, .89E-34.
5. Langage $L_{sep}$ des séparateurs composés de blancs (espace, \t, \n), des commentaires à la C et à la C++.

— Dessiner un unique AFD à jeton reconnaissant une partie de ces langages. Vous reconnaitrez notamment : le mot-clé if, les identificateurs, les entiers décimaux, les flottants sans exposant, les séparateurs. Utiliser des jetons négatifs pour les lexèmes à filtrer (séparateurs).

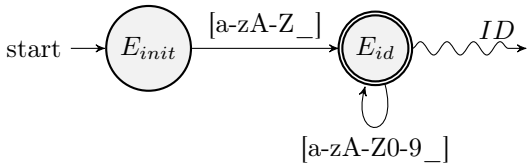
— 1. Automate fini déterministe $L_{key}$ :



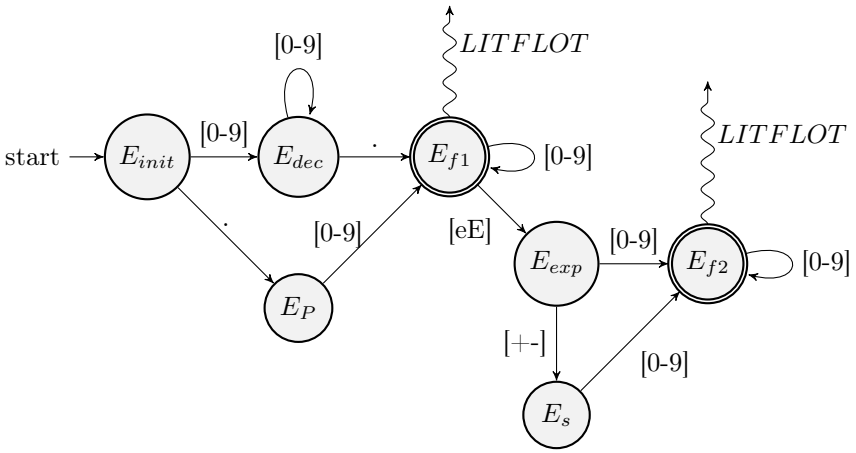
2. Automate fini déterministe de $L_{c10}$, $L_{c8}$ et $L_{c16}$ :



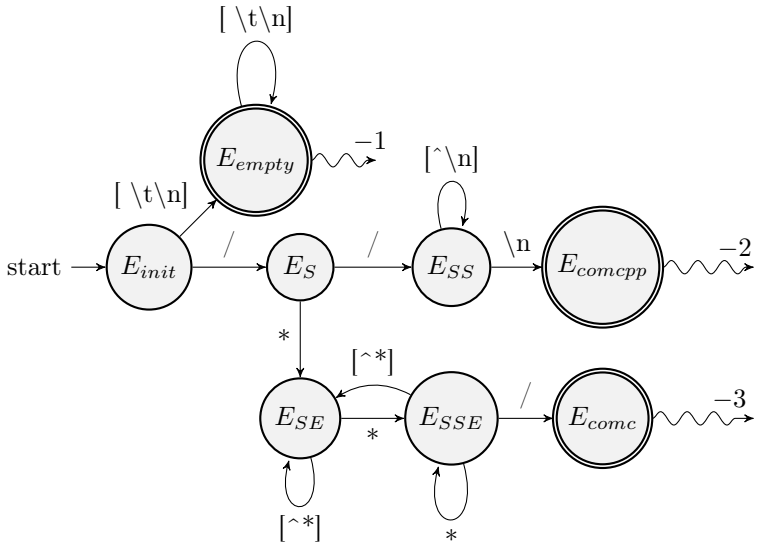

3. Automate fini déterministe de $L_{id}$ :



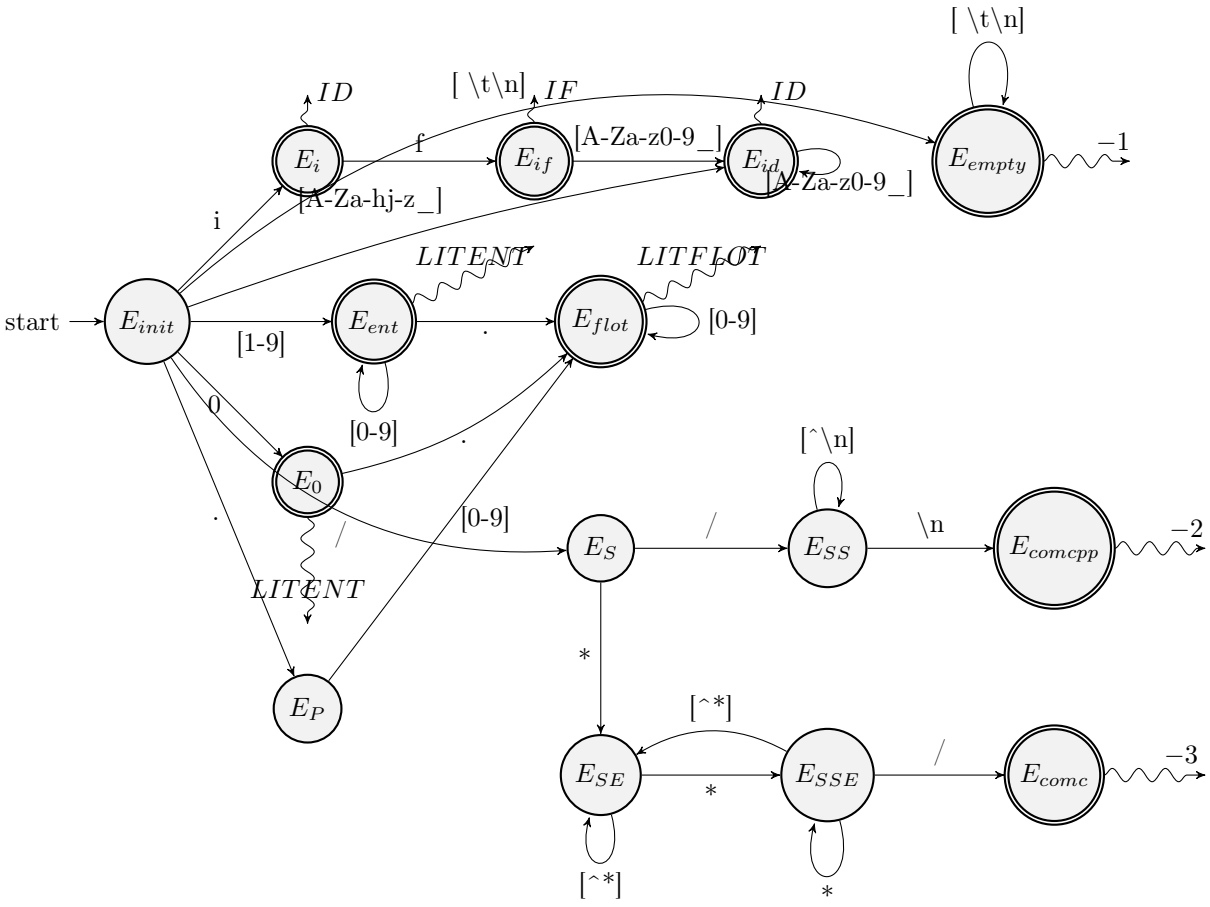

4. Automate fini déterministe de $L_f$ :



5. Automate fini déterministe de $L_{sep}$ :




— AFD de $L_{if}$, $L_{id}$, $L_{c10}$, $L_f$ et $L_{sep}$ :



**Exercice 3 (AFD en C)**

On utilisera dans ce TP, la fonction analex définie dans le cours et qui doit être récupérée sur l'ENT. Trois fichiers sont à télécharger :

- `afd.h` qui contient la définition d'un automate ;
- `analex.h` qui contient la définition de la fonction `analex()` ;
- `analex.c` qui contient un main appelant la fonction `analex()` itérativement ;

L'objectif de ce TP est d'implémenter l'unique AFD à jeton réalisé dans l'exercice précédent de façon à reconnaitre une partie des catégories lexicales du langage C. Pour cela, on modifiera la fonction `creerAfd` du fichier `afd.h`.

1. Lisez et testez les 3 fichiers téléchargés afin d'en comprendre le fonctionnement ;
2. On aura besoin de créer plusieurs transitions d'un état à un autre étiqueté par une classe de caractères comme les minuscules ou les chiffres. Écrire le corps de la fonction suivante :

```c
/** construit un ensemble de transitions de ed à ef pour un intervalle de char
 * @param ed l'état de départ
 * @param ef l'état final
 * @param cd char de début
 * @param cf char de fin
 **/
void classe(int ed, int cd, int cf, int ef);
```

3. En utilisant cette fonction `classe()`, modifiez la fonction `creerAfd()` dans le fichier `afd.h` ;
4. Dans `analex.c`, modifiez le `main()` afin qu'il n'affiche plus d'invite itérativement `analex()` et affichera une chaîne correspondante à l'entier retourné ainsi que le lexème, ceci jusqu'à la fin du fichier.

▶ Voir TP1.

## 1.2 TD/TP 2

**Exercice 4 (expressions régulières)**

Écrire les expressions régulières correspondant aux langages réguliers suivants : $L_{key}$, $L_{c10}$, $L_{c8}$, $L_{c16}$, $L_{id}$, $L_f$, $L_{sep}$.

$L_{key}$ `if|else|th(en|row)`

$L_{c10}$ `([1-9][0-9]*|0)`

$L_{c8}$ `0[0-7]*`

$L_{c16}$ `0[xX][0-9A-Fa-f]*`

$L_{id}$ `[a-zA-Z_][a-zA-Z0-9_]*`

$L_f$ `([0-9]+.[0-9]*|.[0-9]+)([eEdD](\+-)?[0-9]+)?`

$L_{sep}$ `([ \t\n\r\f]|("//".*\n)|("/*"([^*]*\*+[^*/])*[^*]* \*+\/))`

**Exercice 5 (flex)**

Écrire un analyseur lexical reconnaissant l'ensemble des expressions régulières des exercices précédents à l'aide de flex. L'action associée à chaque lexème reconnu consiste à retourner le jeton correspondant au lexème. Toute autre expression d'un caractère retournera un jeton correspondant au code ASCII de ce caractère.

1. Écrivez l'analyseur lexical `analflex.l`
2. Transformez-le en exécutable
3. Testez-le à la ligne de commande puis sur un programme C redirigé. Continuez à améliorer ce programme jusqu'à ce que tous les lexèmes soient reconnus.

Testez cet analyseur

▶ Voir TP2.

**Exercice 6 (supplément)**

Améliorer l'analyseur lexical précédent à l'aide de flex en affectant à une variable globale `yylval` une valeur sémantique dépendant de la catégorie du lexème reconnu :

**mots-clés** rien ;

**LITENT** valeur entière longue (long int) ;

**ID** valeur chaîne de caractères ;

**LITFLOT** flottant double précision ;

▶ Voir TP2.

## 1.3 TD/TP 3

**Exercice 7 (flex nettoyage de texte)**

Écrire un source flex `delblancs.l` filtrant un fichier en :

— supprimant les lignes blanches (lignes vides ou remplies de blancs (espace et tabul.)),

— supprimant les débuts et fins de ligne blancs,

— remplaçant tous les blancs `\t<espace>` multiples par un seul espace,

— remplaçant les tabulations `\t` par un espace.

▶ Voir TP3.

**Exercice 8 (wc en flex)**

Écrire en flex un programme comptant le nombre de lignes, le nombre de mots et le nombre de caractères d'un fichier passé en argument (`man wc`). Un mot est une suite de caractères séparés par des blancs (tab, espaces, retour ligne). Vérifiez que vos résultats sont les mêmes que ceux de `wc`.

▶ Voir TP3.

**Exercice 9 (convertisseur MarkDown (.md) to HTML)**

Markdown est un langage de balisage léger destiné à offrir une syntaxe facile à lire et à écrire utilisée par GitHub et GitLab (readme.md). Un document balisé par Markdown peut être converti facilement en HTML. Pour réaliser des listes non numérotées (`<ul><li>`) :

— une ligne vide précède et suit la liste de 1er niveau

— chaque item est précédé d'une étoile (*) ou d'un tiret (-) en début de ligne

— on peut emboîter des listes en indentant avec au moins 2 espaces de plus que le niveau parent.

L'exemple suivant :

```
* 1
    * 1, 1
        - 1, 1, 1
            * 1, 1, 1, 1
            * 1, 1, 1, 2
    - 1, 2
* 2
```

produira :

```
<ul><li>1</li>
<ul><li>1, 1</li>
<ul><li>1, 1, 1</li>
<ul><li>1, 1, 1, 1</li>
<li>1, 1, 1, 2</li>
</ul></ul><li>1, 2</li>
</ul><li>2</li>
</ul>
```

Écrire le source flex permettant d'effectuer la traduction.

▶ Voir TP3.

## 1.4 TD/TP 4

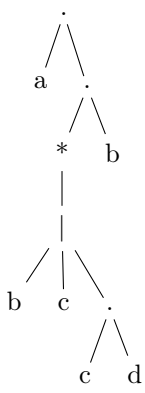
**Exercice 10 (algorithmique des automates d'états finis)**

Soit l'expression régulière $e$ suivante : $e = a((b|c)^{**}|cd)^*b$

1. Dessiner un automate fini équivalent à $e$.
2. Cet automate est-il déterministe ? Si oui, indiquez pour quelle raisons, sinon, déterminisez-le.
3. Minimisez cet AFD.

1. Pour commencer, simplifions l'expression régulière $e$. On sait que $r^* = r^n$ donc $r^{**} = r^{n^m} = r^{nm} = r^*$. Ainsi, on a $(b|c)^{**} = (b|c)^*$. De plus, $(r*|s)^* = r^{**}|s* = r^*|s^* = (r|s)^*$ donc $((b|c)^*|cd)^* = (b|c|cd)^*$. On peut réécrire $e$ en $a(b|c|cd)^*b$.

   Appliquons la méthode de Thompson pour construire l'automate fini équivalent à $e$. On construit tout d'abord l'arbre de syntaxe abstrait de l'expression régulière :

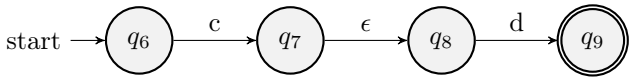



Dans l'arbre ci-contre, l'opération de concaténation a été associée à droite, mais on peut aussi construire l'arbre qui associe la concaténation à gauche. Celui-ci changerait, mais l'automate fini qui en résulte ne changerait pas.

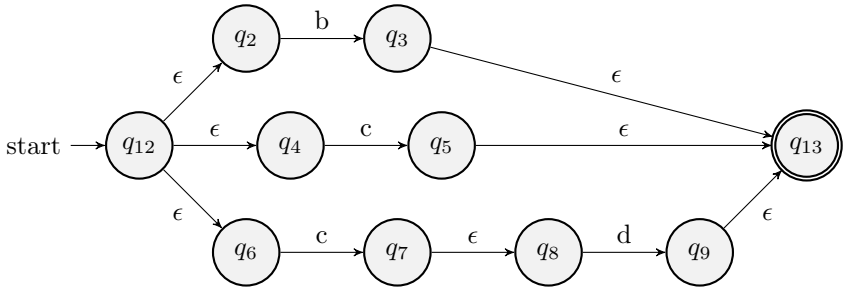
   On peut maintenant construire itérativement l'automate fini équivalent à $e$, en commençant par les feuilles, et en remontant tout l'arbre :

   Étape 1 : création des noeuds des feuilles (de gauche à droite) :

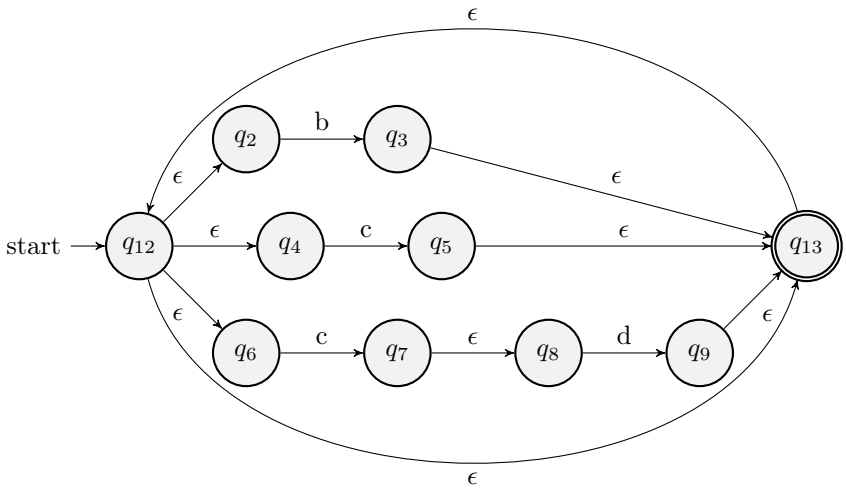

   Étape 2 : concaténation de $c$ et $d$ :



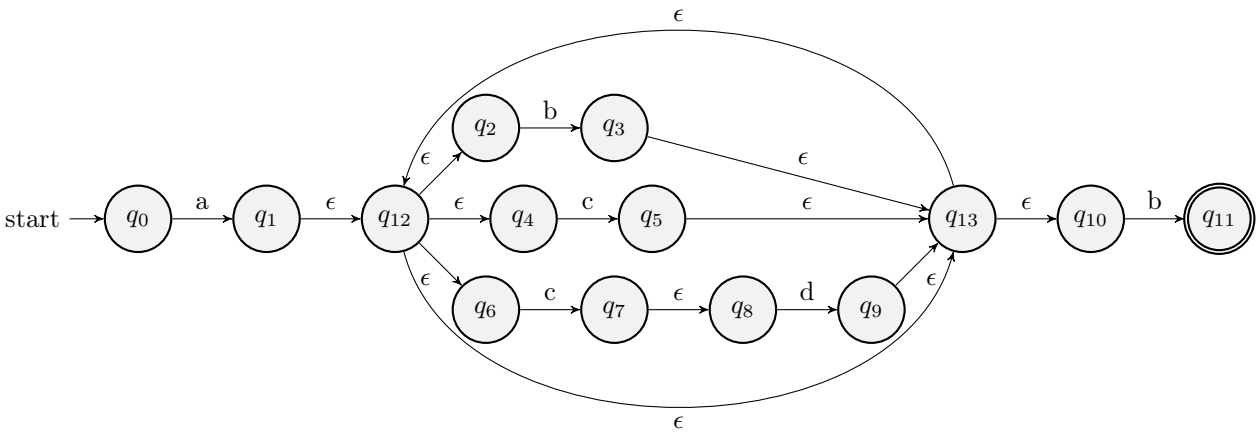

   Étape 3 : application du | sur les feuilles $b$, $c$ et $cd$ :




Étape 4 : application de l'étoile de Kleene :




Étape 5 : concaténation des deux feuilles restantes :

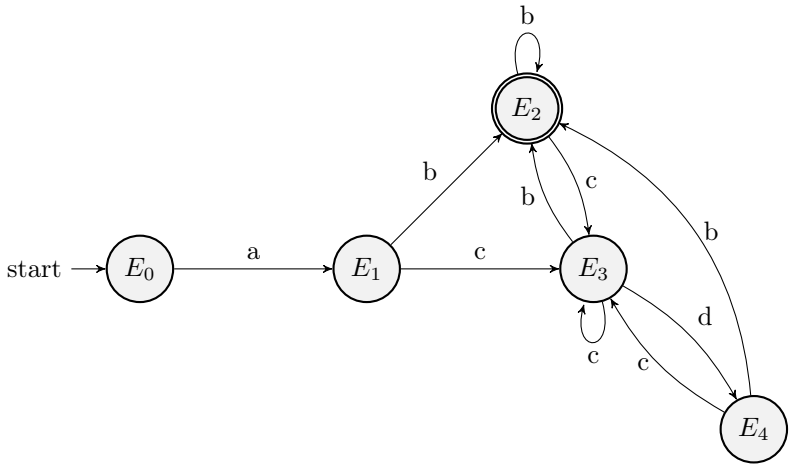



On a construit un automate fini non déterministe (car $q_{12}$ va vers 3 états différents avec la même transition) équivalent à $e$.

2. Pour le déterminiser, il faut supprimer toutes les $\epsilon$-transitions. Pour ce faire, on va déterminer les différents états de l'automate :
   - Depuis $q_0$, les états atteignables avec $a$ sont : $q_1, q_{12}, q_2, q_4, q_6, q_{13}, q_{10}$.
   - Depuis ces derniers, on peut atteindre :
     - $q_2, q_3, q_4, q_6, q_{10}, q_{11}, q_{12}, q_{13}$ avec $b$,
     - $q_2, q_4, q_5, q_6, q_7, q_8, q_{10}, q_{12}, q_{13}$ avec $c$.
   - Si on a $ab$, on peut atteindre :
     - $q_2, q_3, q_4, q_6, q_{10}, q_{11}, q_{12}, q_{13}$ avec $b$. On remarque que ce sont les mêmes que les précédents, il y a donc une boucle.
     - $q_2, q_4, q_5, q_6, q_7, q_8, q_{10}, q_{12}, q_{13}$ avec $c$. On remarque que ce sont les mêmes que les précédents, on pourra donc relier le noeud avec le précédent.
   - Si on a $ac$, on peut atteindre :
     - $q_2, q_3, q_4, q_6, q_{10}, q_{11}, q_{12}, q_{13}$ avec $b$. On remarque que ce sont les mêmes que les précédents, on pourra donc relier le noeud avec celui-ci.
     - $q_2, q_4, q_5, q_6, q_7, q_8, q_{10}, q_{12}, q_{13}$ avec $c$. On remarque que ce sont les mêmes que les précédents, on pourra donc relier le noeud avec le précédent.
     - $q_2, q_4, q_6, q_9, q_{10}, q_{13}, q_{12}$ avec $d$.
   - Si on a $acd$, on peut atteindre :
     - $q_2, q_3, q_4, q_6, q_{10}, q_{11}, q_{12}, q_{13}$ avec $b$. On remarque que ce sont les mêmes que les précédents, on pourra donc relier le noeud avec celui-ci.
     - $q_2, q_4, q_5, q_6, q_7, q_8, q_{10}, q_{12}, q_{13}$ avec $c$. On remarque que ce sont les mêmes que les précédents, on pourra donc relier le noeud avec le précédent.

   On trouve 5 états différents en déterminisant cet algorithme :
   - $E_0$, l'état de départ,
   - $E_1 = q_1, q_{12}, q_2, q_4, q_6, q_{13}, q_{10}$ le noeud $a$,
   - $E_2 = q_2, q_3, q_4, q_6, q_{10}, q_{11}, q_{12}, q_{13}$ l'état final,
   - $E_3 = q_2, q_4, q_5, q_6, q_7, q_8, q_{10}, q_{12}, q_{13}$ l'état où $c$ boucle,
   - $E_4 = q_2, q_4, q_6, q_9, q_{10}, q_{12}, q_{13}$ le noeud avec $d$.

   On construit l'automate fini déterministe suivant :



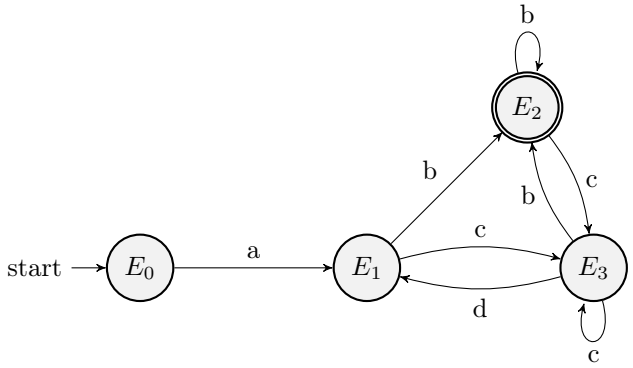

3. Pour savoir si l'AFD précédent est minimisable, on pose les transitions suivantes :

— $E_0 : a$,
— $E_1 : b, c$,
— $E_2 : b, c$,
— $E_3 : b, c, d$,
— $E_4 : b, c$.

On peut ensuite catégoriser chaque état :

— $\boxed{E_0, E_1, E_3, E_4, p \,\|\, E_2}$ $E_2$ se distingue des autres car c'est l'état final. $p$ est le puit qui a été ajouté pour reproduire un automate complet.
— $\boxed{E_0 \mid E_1, E_3, E_4, p \,\|\, E_2}$ on distingue $E_0$ par $a$.
— $\boxed{E_0 \mid E_3 \mid E_1, E_4, p \,\|\, E_2}$ on distingue $E_3$ par $d$.
— $\boxed{E_0 \mid E_3 \mid E_1, E_4 \mid p \,\|\, E_2}$ on distingue $E_1$ et $E_4$ par $b$.

On ne peut pas distinguer $E_1$ et $E_4$. Ces deux états peuvent donc être fusionnés, pour nous donner l'AFD minimal suivant :

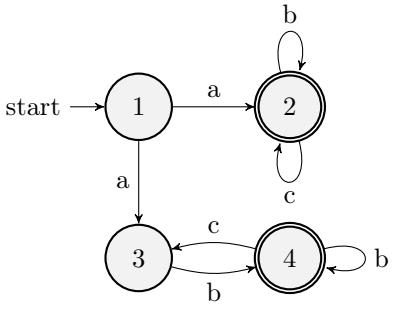


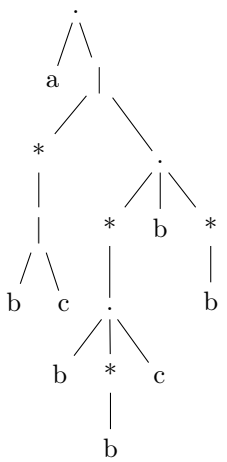

**Exercice 11**

Soit l'automate fini $B = (\{a, b, c\}, \{1, 2, 3, 4\}, \{1\}, \{2, 4\}, \{1a2, 1a3, 2b2, 2c2, 3b4, 4c3, 4b4\})$.

1. Dessiner un automate fini déterministe équivalent à $B$.
2. Minimisez cet AFD.

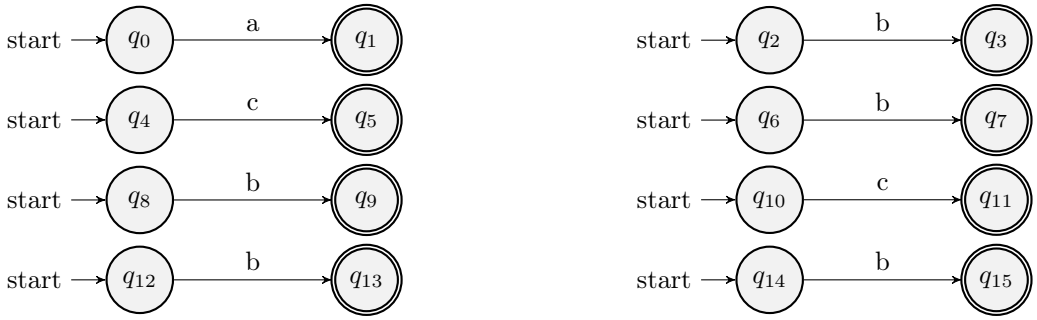
1. On peut construire l'automate fini suivant (à gauche) avec la définition de $B$ :



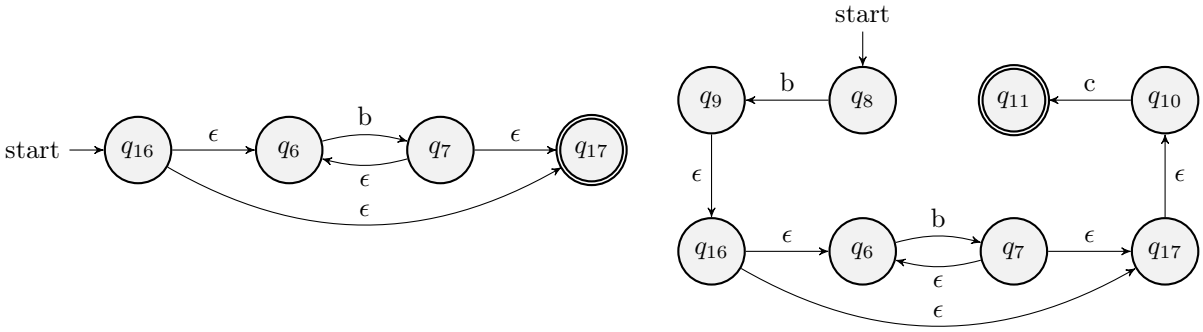



C'est un automate fini non déterministe, qui reconnait l'expression régulière $e = a((b|c)^*|(bb^*c)^*bb^*)$. L'arbre syntaxique de cette expression régulière est à droite de l'automate. En suivant le même procédé que tout à l'heure, on peut normaliser cet automate puis le déterminiser.

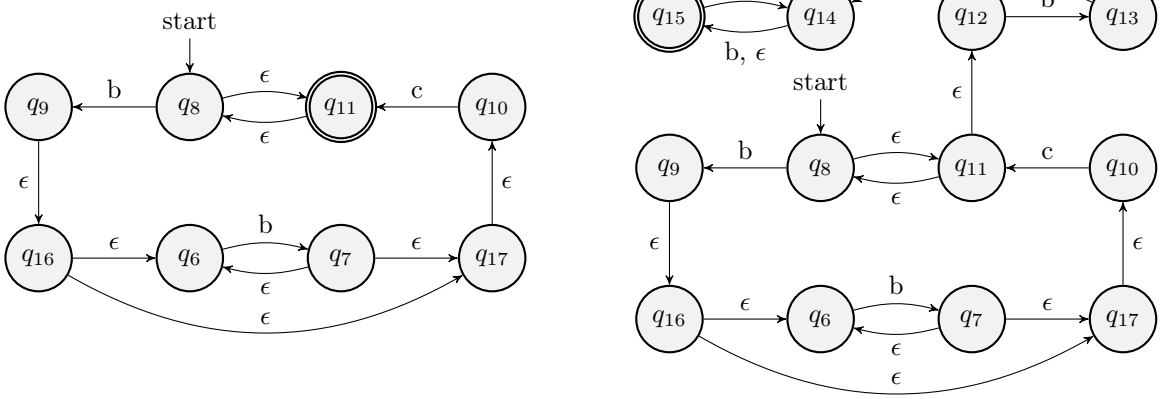
Étape 1 : création des noeuds des feuilles (de gauche à droite) :



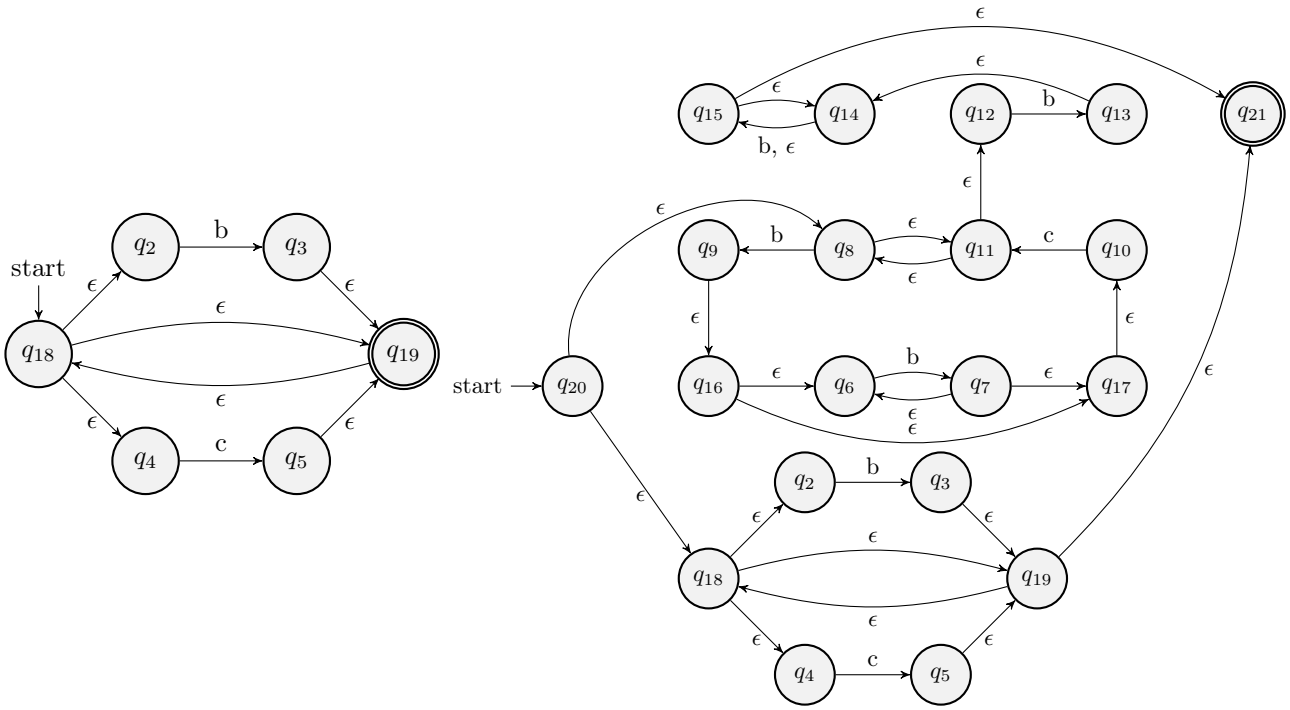

Étape 2 : création de $b^*$ et concaténation de $bb^*c$ :



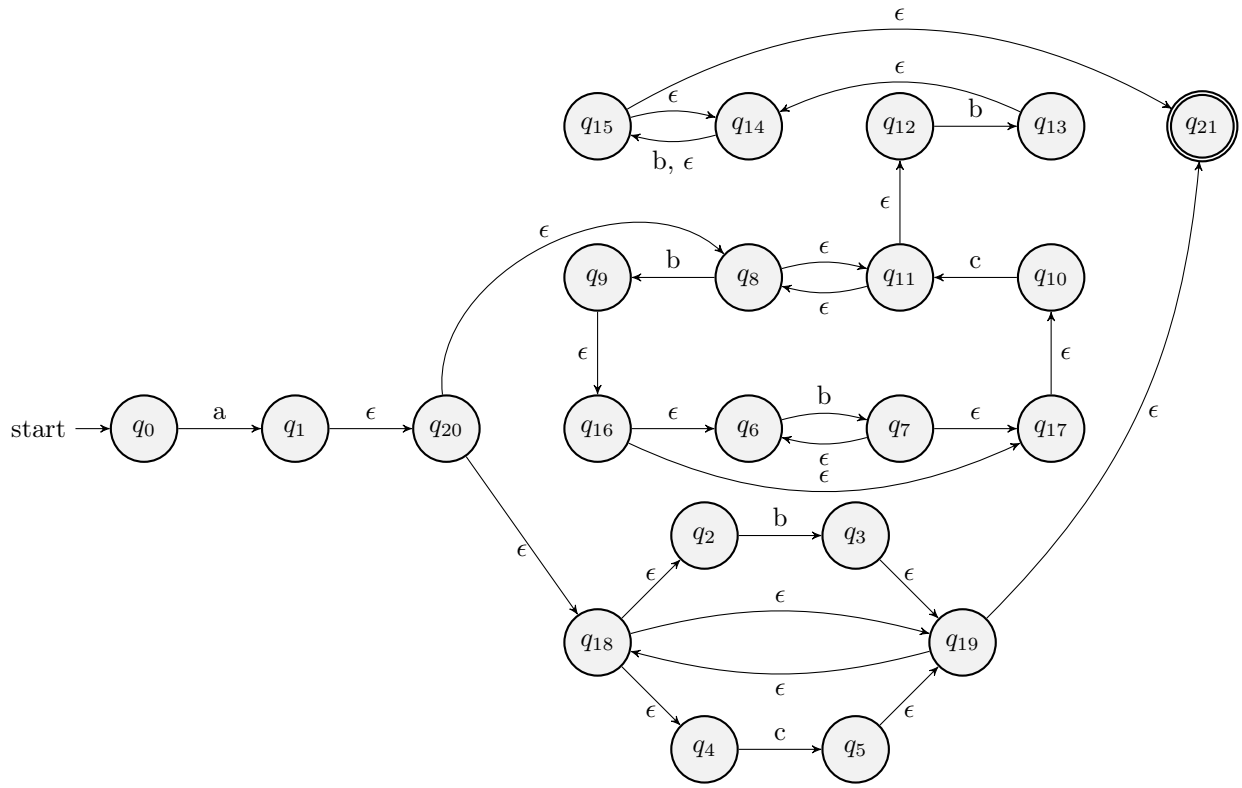

Étape 3 : création de $(bb^*c)^*$ et concaténation à $bb^*$ pour donner $(bb^*c)^*bb^*$ :




Étape 4 : $(b|c)^*$ et choix entre $(b|c)^*$ et $(bb^*c)^*bb^*$ :

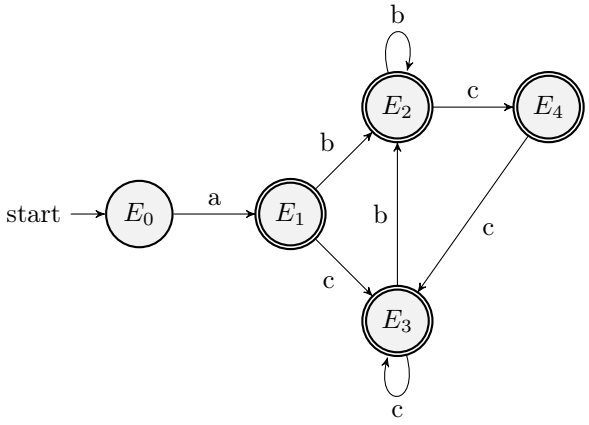



Dernière étape : concaténation avec a :




On a construit l'AFN depuis l'expression régulière décrite par $B$. Il n'y a plus qu'à appliquer l'algorithme

de déterminisation pour avoir un AFD équivalent à $B$ :




Pour le réaliser, on a remarqué qu'il y a 5 états différents :

— $E_0 = \{q_0\}$,
— $E_1 = \{q_1, q_2, q_4, q_8, q_{11}, q_{12}, q_{18}, q_{19}, q_{20}, q_{21}\}$,
— $E_2 = \{q_2, q_3, q_4, q_6, q_7, q_8, q_{10}, q_{12}, q_{13}, q_{14}, q_{15}, q_{16}, q_{17}, q_{18}, q_{19}, q_{21}\}$,
— $E_3 = \{q_2, q_4, q_5, q_{18}, q_{19}, q_{21}\}$,
— $E_4 = \{q_2, q_4, q_5, q_8, q_{11}, q_{12}, q_{18}, q_{19}, q_{21}\}$.

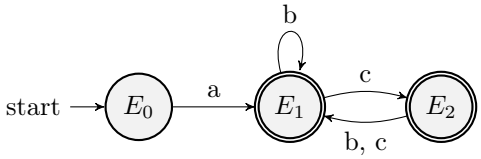
2. On pose les transitions suivantes :

— $E_0 : a$,
— $E_1 : b, c$,
— $E_2 : b, c$,
— $E_3 : b, c$,
— $E_4 : c$.

On peut ensuite catégoriser chaque état :

— $\boxed{E_0, p \,\|\, E_1, E_2, E_3, E_4}$
— $\boxed{E_0 \mid p \,\|\, E_1, E_2, E_3, E_4}$ on distingue $E_0$ de $p$ par $a$.
— $\boxed{E_0 \mid p \,\|\, E_1, E_2, E_3 \mid E_4}$ on distingue $E_1, E_2, E_3$ de $E_4$ par $b$.

On ne peut pas distinguer $E_1, E_2$ et $E_3$. Ces trois états peuvent donc être fusionnés, pour nous donner l'AFD minimal suivant :




**Exercice 12 (amélioration des TPs précédents)**

Vous pouvez reprendre les exercices précédents afin d'améliorer la reconnaissance des jetons du langage C.

▶ Voir TP3.

# 2 Analyse descendante récursive

## 2.1 TD/TP 5

**Exercice 13 (analyse descendante récurisve)**

Soit la grammaire non récursive à gauche vue en cours $G_{ENR} = (\{0, 1, \ldots, 9, +, *, (, )\}, \{E, R, T, S, F\}, X, E)$ avec les règles $X$ suivantes :

$$E \to TR$$
$$R \to +TR|\varepsilon$$
$$T \to FS$$
$$S \to *FS|\varepsilon$$
$$F \to (E)|0|1|\ldots|9$$

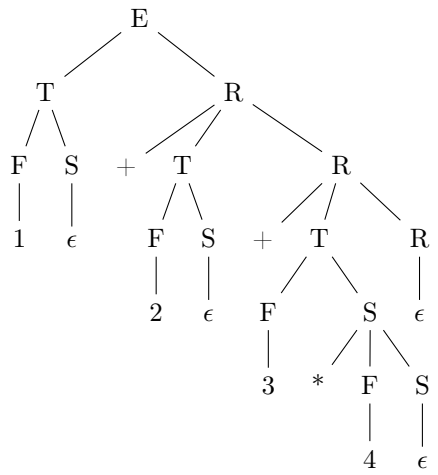
Soit le programme C vérifiant un mot du langage $L(G_{ENR})$ : `analdesc.c`.

1. Dessiner l'arbre de dérivation associé au mot : `1+2+3*4`. Dessiner l'arbre des appels récursifs des fonctions E, R, T, S, F lorsqu'on reconnaît ce même mot. Que remarquez-vous ?
2. On souhaite implémenter l'évaluation de la valeur d'une expression arithmétique pendant sa vérification syntaxique. La multiplication sera prioritaire par rapport à l'addition et l'associativité des deux opérateurs sera à gauche (de gauche à droite). Annoter l'arbre obtenu à la question précédente pour indiquer où sont effectuées les 3 opérations et comment les fonctions transmettent leurs résultats.
3. Sur le modèle du vérificateur, écrire un programme évaluant la valeur d'une expression arithmétique. Par exemple, **evaldesc** sur la chaîne `1+2+(2+1)*3` retournera 12. Attention, on précisera pour chaque opérateur le type d'associativité, gauche ou droite, employée dans le programme.
4. Sur le modèle du vérificateur, écrire un programme traduisant une expression arithmétique en sa forme postfixée (polonaise inversée). Par exemple, **postdesc** sur la chaîne `1+2+(2+1)*3` retournera `12+21+3*+`. On utilisera l'associativité à gauche pour l'addition et la multiplication.
5. Sur le modèle du vérificateur, et en utilisant le type abstrait `Arbin` implémenté en C, écrire un analyseur syntaxique produisant et affichant l'arbre abstrait correspondant à une expression. On utilisera l'associativité à gauche pour l'addition et la multiplication. Par exemple, **arbindesc** sur la chaîne `1+2+(2+1)*3` affichera :

```
+
    +
        1
        2
    *
        +
            2
            1
        3
```

▶ Voir TP4 (pour les questions 3, 4 et 5).

1. L'arbre de dérivation et l'arbre des appels récursif est plus ou moins similaire :




On remarque que la priorité des opérations est respectée d'emblée dans cet arbre.

2. Il suffit de faire renvoyer des entiers à toutes les fonctions, et à faire prendre un paramètre entier à R et S (pour que ces fonctions puissent effectuer le calcul dans leur corps).