Correction des exercices de TD

Table des matières

1 Analyse Lexicale

1.1 TD1

Exercice 1

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 `;`).

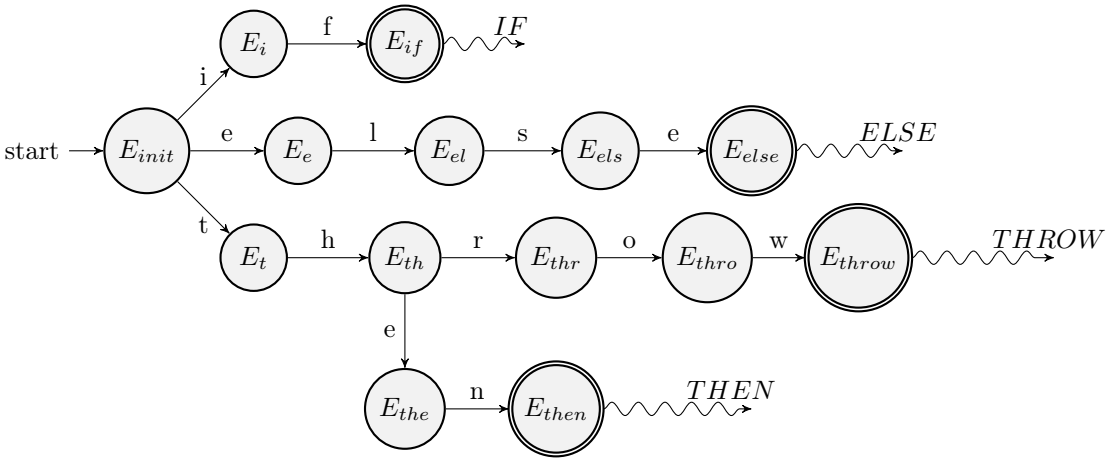
Exercice 2

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

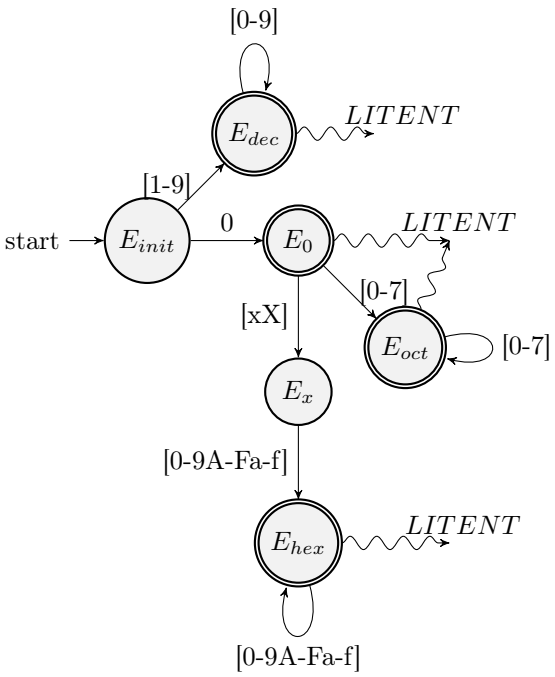
1. Langage de dertains mots-clés du C : $L_{key} = \{if, then, else, throw\}$ (sensible à la casse).
2. Langage des littéraux numériques entiers du C (ou C++, ou Java), décimaux L_{c10} , octaux L_{c8}, hexadécimaux L_{c16}.
3. Langage L_{id} des identificateurs composés d'une lettre au moins, éventuellement 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).

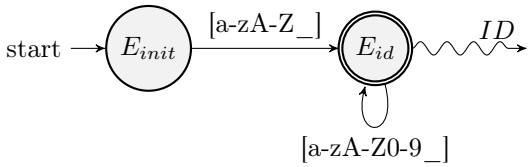
— 1. Automate fini déterministe L_{key} :



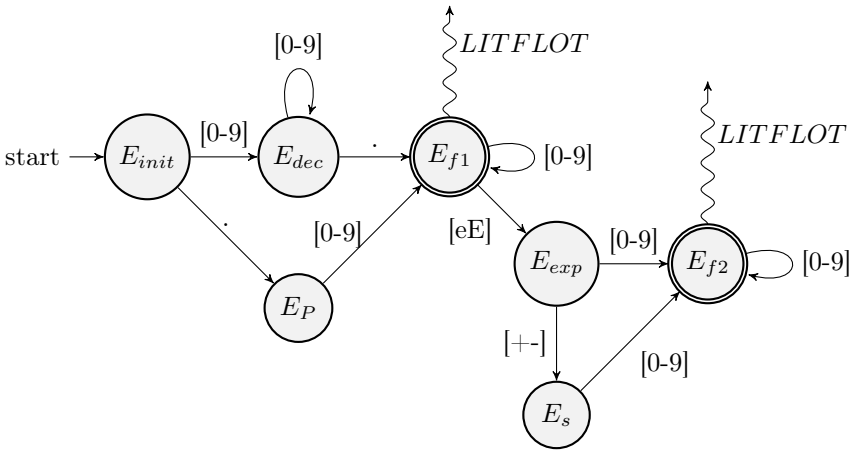
2. Automate fini déterministe de L_{c10}, L_{c8} et L_{c16} :



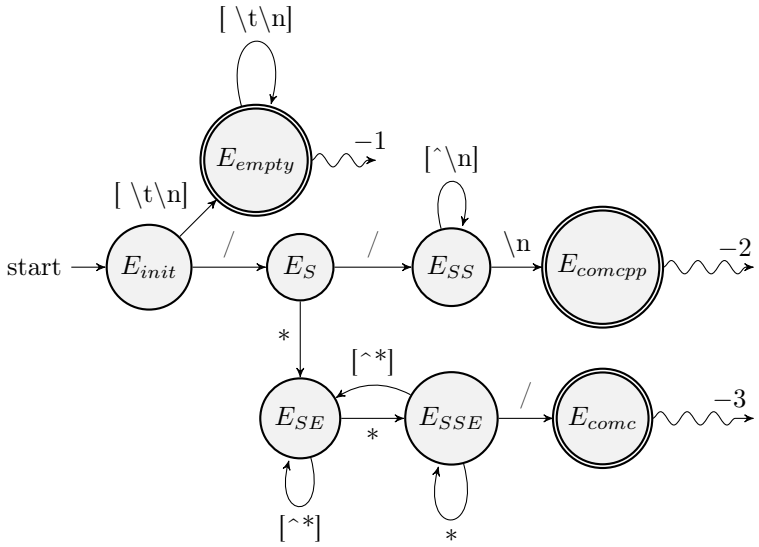
3. Automate fini déterministe de L_{id} :



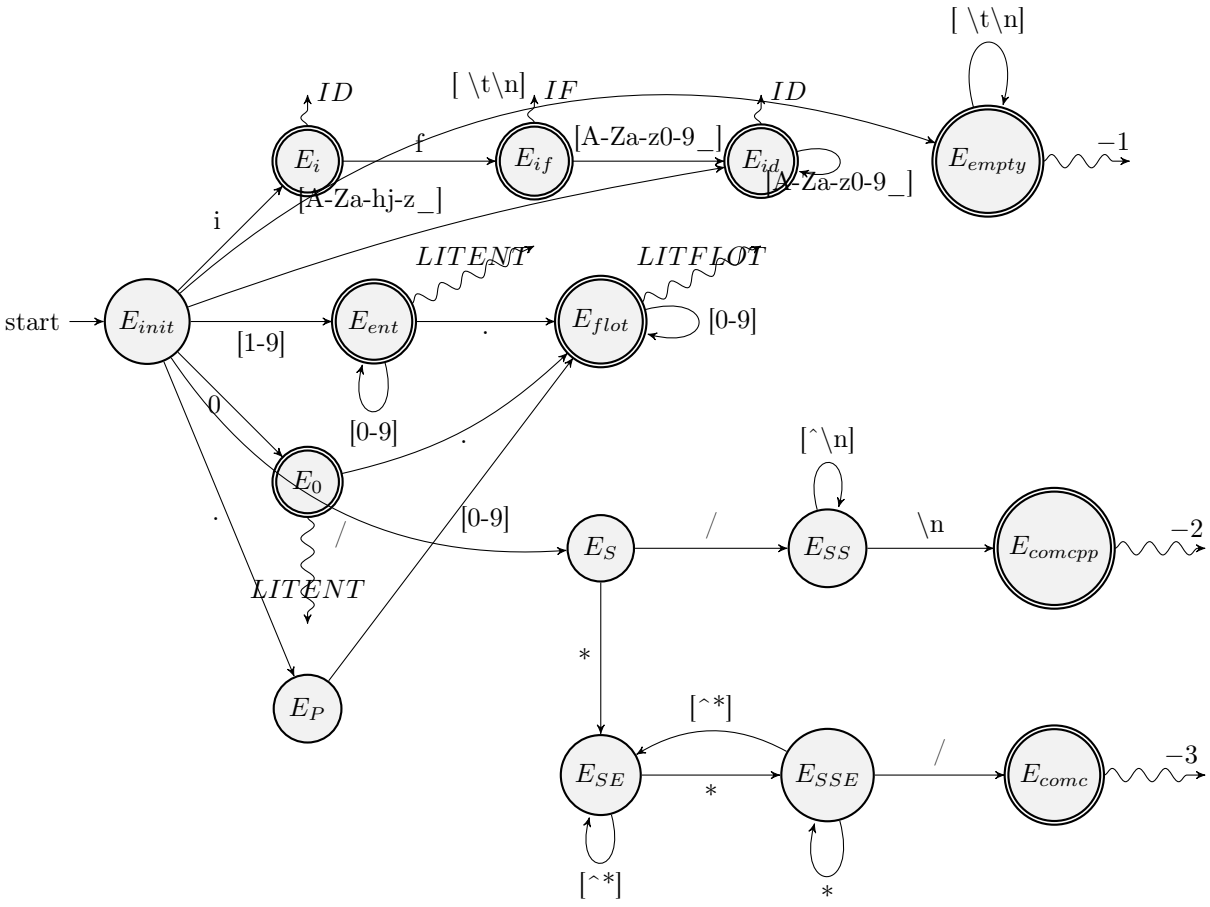
4. Automate fini déterministe de L_f :



5. Automate fini déterministe de L_{sep} :



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



Exercice 3

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 TD2

Exercice 4

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

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

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.