Devoir Surveillé

Semestre : 1 ■ 2 □

Session : Principale ■ Rattrapage □

Module :Théorie des Langages

Enseignant(s) : Equipe TLA Classe(s) : 3A33-3A65

Documents autorisés : OUI □ NON ■

Calculatrice autorisée : OUI □ NON ■ Internet autorisée : OUI □ NON ■

Date : 31/10/2024 Heure : 11h00 Durée :1h Nombre de pages : 2

Les copies soignées sont toujours appréciées

Exercice 1 : (8 pts)

Soit l'expression régulière suivante : $\mathbf{E_R}$ = ((a|b)(a|b))*(a|b)

Partie1 :

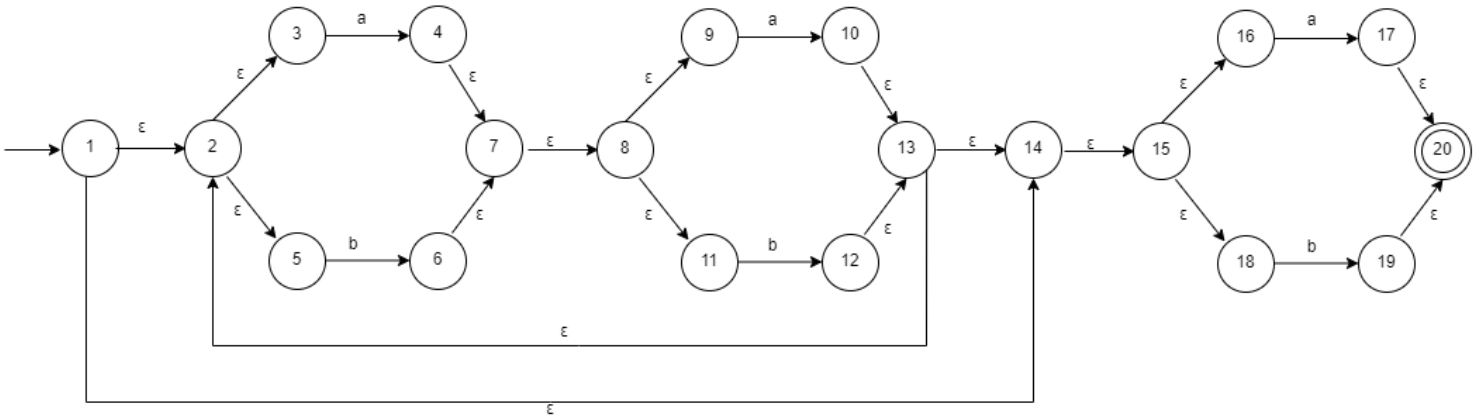
1. Décrivez en une phrase le langage reconnu par $\mathbf{E_R}$. (**0.5 pt**)

   L' $\mathbf{E_R}$ reconnait tous les mots de taille impaire sur $\Sigma = \{a,b\}$

2. Écrivez une description formelle du langage reconnu. (**0.5 pt**)

   $L_1 = \{w \in \{a,b\}^* / |w| = 2k+1, k \geq 0\}$

3. Utilisez l'algorithme de Thompson pour construire un automate fini non déterministe (AFN) qui reconnaît le langage décrit par l'expression régulière $\mathbf{E_R}$. (**2 pt**)

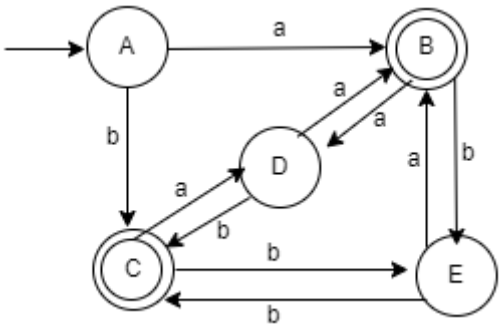


Partie2 :

1. À partir de l'automate non déterministe que vous avez construit, effectuez la déterminisation pour obtenir un automate fini déterministe (AFD) équivalent. (**2pts**)

1	{1,2, 3,5,14,15,16,18}	2	{2,3,5}
3	{3}	4	{4,7,8,9,11}
5	{5}	6	{6,7,8,9,11}
7	{7,8,9,11}	8	{8,9,11}
9	{9}	10	{10,13,14,15,16,18,2,3,5}
11	{11}	12	{12,13,14,15,16,18,2,3,5}
13	{13,14,15,16,18,2,3,5}	14	{14,15,16,18}
15	{15,16,18}	16	{16}
17	{17,20}	18	{18}
19	{19,20}	20	{20}

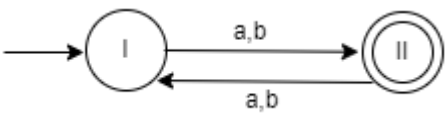
	a	b
A={1,2, 3,5,14,15,16,18}	{4,7,8,9,11,17,20} = B	{6,7,8,9,11,19,20}=C
B	{10,13,14,15,16,18,2,3,5}=D	{12,13,14,15,16,18,2,3,5}= E
C	D	E
D	B	C
E	B	C



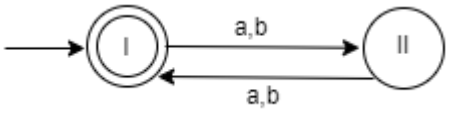
2. Minimisez l'automate obtenu à la question précédente. (**2pts**)

I II

(A D E) (B C)
II II II I I
II II II I I



3. **Déduisez un automate B** qui reconnaît les mots de taille paire. (**1pt**)



Exercice 2 : (6 pts)

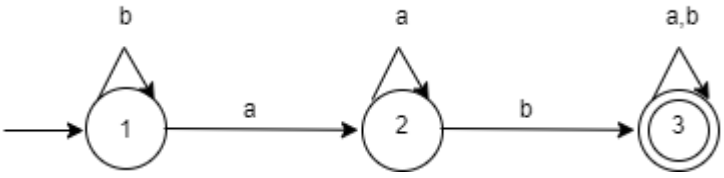
Considérez les langages suivants :

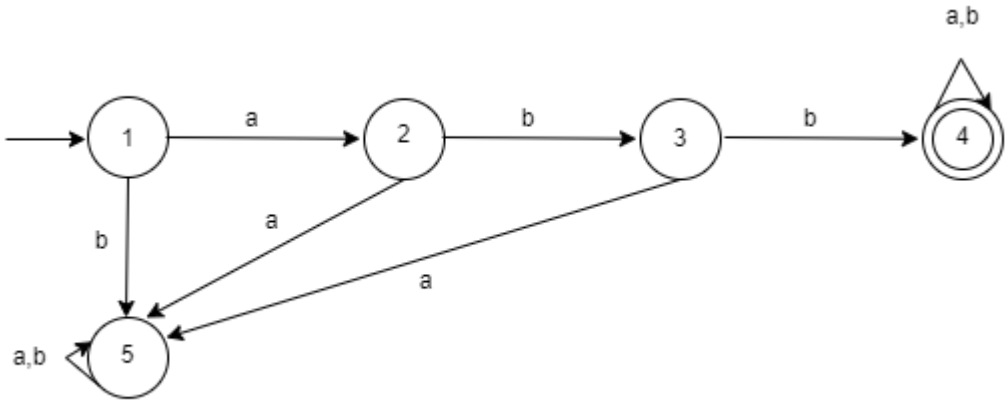
L_1 = {w ∈ {a,b}* | w contient ab}

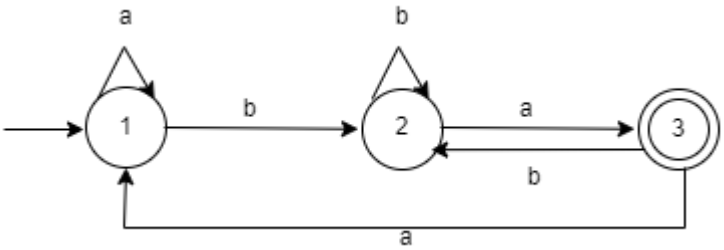
L_2 = {w ∈ {a,b}* | w commence par abb}

L_3 = {w ∈ {a,b}* | w se termine par ba}

1. Construisez directement les automates finis déterministes (AFD) qui reconnaissent les langages L_1, L_2, et L_3. **(3pts)**







2. Donnez, pour chacun de ces langages, une expression régulière représentant son complément. (**1.5pt**)

E_{1C}= b*a*

E_{2C}=(ε|b|aa|aba)(a|b)*

E_{3C}=ε| a | (a|b)*b |(a|b)*aa

3. Trouvez les plus petit mots suivants, si possible:

a) $w_1 \in L_1 \cap L_2 \cap L_3$ b) $w_2 \in L_2$ et $w_2 \notin L_1$ c) $w_3 \notin L_1 \cup L_2 \cup L_3$ **(1.5pt)**

a) w_1=abba b) $\emptyset$ c) $w_3 = \varepsilon$

Exercice 3 (6 pts)

On souhaite définir un analyseur lexical à l'aide de Flex pour un langage de programmation simple qui reconnaît les éléments suivants :

- **Les variables** : Elles commencent par une lettre et sont composées de lettres et de chiffres (les noms de variables sont sensibles à la casse)
- **Les opérateurs arithmétiques** : +, -, *, /
- **Les parenthèses** : (,)
- **Les mots-clés** : `print`, `let`, `if`, `then`
- **Les entiers** : Ils sont composés uniquement de chiffres (0-9)

1. Compléter les parties manquantes du fichier de spécification Flex suivant pour construire un analyseur lexical reconnaissant les éléments mentionnés ci-dessus et permettant de retourner sur la console, à chaque identification d'un lexème, la chaine reconnue ainsi que la description. **(3 pts)**

```
%{
   #include <stdio.h>
   #include <stdlib.h>
%}
%%
   /* Variables (sensible à la casse) */
   /* Opérateurs arithmétiques */
   /* Parenthèses */
   /* Mots-clés */
   /* Les entiers */

…… printf(……);
…… printf(……);
"print"        { printf("MOT-CLÉ: print\n"); }
"let"          { printf("MOT-CLÉ: let\n"); }
"if"           { printf("MOT-CLÉ: if\n"); }
"then"         { printf("MOT-CLÉ: then\n"); }
a-zA-Z][a-zA-Z0-9]* { printf("VARIABLE: %s\n", yytext);
}
"+" { printf("OPÉRATEUR: +\n"); }
"-" { printf("OPÉRATEUR: -\n"); }
"*" { printf("OPÉRATEUR: *\n"); }
"/" { printf("OPÉRATEUR: /\n"); }

"(" { printf("PARENTHESE: (\n"); }
")" { printf("PARENTHESE: )\n"); }
```

```
    [0-9]+ { printf("ENTIER: %s\n", yytext); }
    %%
    int main(){
    yylex() ;
    return 0 ; }
```

2. Expliquez pourquoi l'ordre des règles dans Flex est important. Analysez les règles que vous avez définies précédemment et ajustez leur ordre si nécessaire pour assurer une reconnaissance correcte des éléments. **(1pt)**

   L'ordre des règles dans Flex est crucial car Flex fonctionne selon une approche de "premier match", ce qui signifie que lorsqu'il parcourt l'entrée, il applique la première règle qui correspond. Ainsi, si une règle plus générale est placée avant une règle plus spécifique, la règle générale sera appliquée, ce qui peut empêcher la règle plus spécifique de jamais être utilisée.

   Dans notre exemple, les règles pour les mots réservés doivent être placées avant celles des identificateurs, pour éviter que des mots réservés soient reconnus à tort comme des identificateurs.

3. Que se passe-t-il si une règle n'est pas définie pour un symbole rencontré dans le texte d'entrée ? Proposez une solution pour gérer ce type de situation. **(1pt)**

   Si une règle n'est pas définie pour un symbole rencontré dans le texte d'entrée dans Flex, ce symbole est ignoré par défaut.

   Ajouter une règle à la fin du fichier Flex qui capture tous les symboles non reconnus (`.* printf("Non Défini");` par exemple)

4. Analysez la chaîne suivante : `let x = 10 + y;` Quel est le résultat de cette analyse en termes de tokens ? **(1pt)**

   Voici les tokens identifiés par l'analyseur lexical :

   - **let** : MOT-CLÉ
   - **x** : VARIABLE
   - **=** : Non défini
   - **10** : ENTIER
   - **+** : OPÉRATEUR
   - **y** : VARIABLE
   - **;** : Non défini

Bon Travail