Chapitre II.
Analyse lexicale

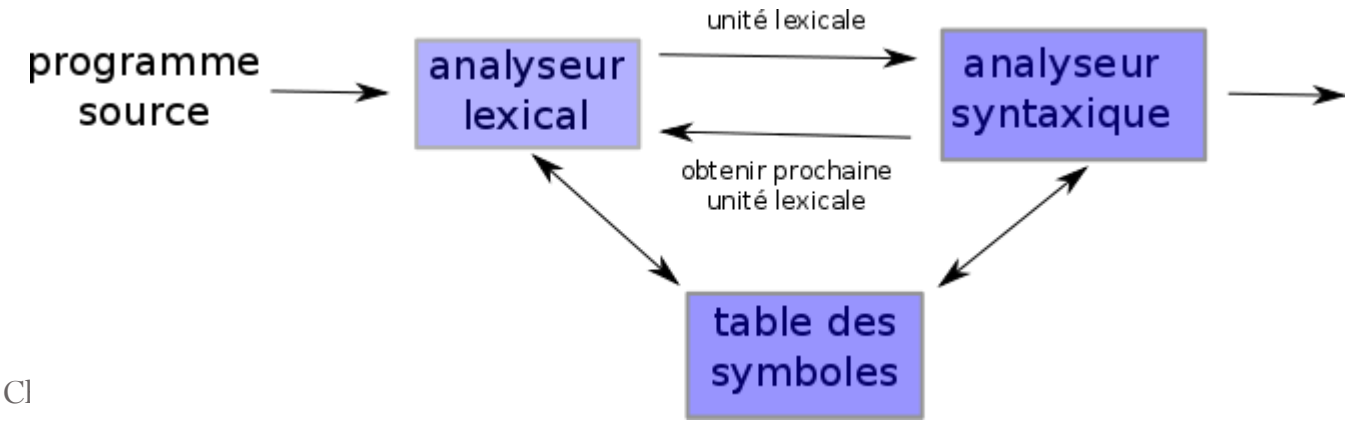
a.hettab@centre-univ-mila.dz

Introduction

- L'analyse lexicale consiste à transformer d'un ensemble de caractères du programme source en **unités lexicales** afin de les fournir à **l'analyseur syntaxique**.
- Les unités lexicales les plus connues sont:
  - Mots clefs (for, if,. . .)
  - Symboles (+, , (,), . . .)
  - Identificateurs (toute suite de lettres et de chiffres qui commence par une lettre).
  - Nombres (toute suite de chiffres).
- les commentaires et les blancs sont ignorés dans la phase d'analyse lexicale.

Introduction

- Dans l'analyse lexicale, on distingue les concepts suivants :
- **Unité lexicale** : correspond a une entité (un concept) renvoyé par l'analyseur lexical. Par exemple <; >;<=;>= sont tous des operateurs relationnels.
- **Un lexème** est une instance d'unité lexicale. Par exemple le lexème 6,28 une instance de l'unité lexicale nombre.
- **Un modèle** associe des lexèmes a leur unité lexicale correspondante.



Les langages formels

Définitions:

- On se donne un ensemble Σ appelé alphabet, dont les éléments sont appelés caractères.
- Un mot (sur Σ) est une séquence de caractères (de Σ). On note :
  - ε le mot vide,
  - ***uv*** la concaténation des mots *u* et *v* (la concaténation est associative et ε est un élément neutre).
  - Σ * est l'ensemble des mots sur Σ.
  - Σ + est l'ensemble des mots non vides sur Σ.

- Un langage sur Σ est un sous-ensemble L de Σ *.

Les langages formels

Exemples:

- Σ_1 est l'alphabet et L_1 est l'ensemble des mots du dictionnaire français avec toutes leurs variations (pluriels, conjugaisons). L_2 est l'ensemble des phrases grammaticalement correctes de la langue française.
- Σ_2 est l'ensemble des caractères **ASCII**, et L_3 est composé de tous les mots clés de pseudo pascal: des symboles, des identificateurs, et de l'ensemble des entiers décimaux…. et L_4 est l'ensemble des programmes pseudo pascal.
- Σ_3 est $\{a, b\}$ et L_5 est $\{ a^n b^n \mid n \in IN\}$ (tous les mots qui constituent de a et b et la langueur de a = la langueur de b).

Les expressions régulières

- **Les expressions régulières** représentent un formalisme simple permettant de décrire certains langages simples (**les langages réguliers**).
- Elles permettent de décrire les **unités lexicales** d'une manière concise et compacte.
- Le générateur d'analyseur lexical **LEX** utilise les expressions régulières pour spécifier ses unités lexicales.

Les expressions régulières

Définition:

- On note a, b, etc. des lettres de l'alphabet Σ. M et N sont des expressions régulières et $L[M]$ le langage associé à M.
- Une lettre a désigne le langage $\{a\}$.
- Epsilon: ε désigne le langage $\{\varepsilon\}$.
- Concaténation: $M\ N$ désigne le langage $L[M]] \cap L[N]$.
- Alternative: $M \mid N$ désigne le langage $L[M] \cup L[N]$.
- Répétition: M^* désigne le langage $(L[M])^*$.
- $M?$ pour $M \mid \varepsilon$ et M^+ pour $M\ M^*$.

Les expressions régulières

Exemples:

- **Lettre:** `[A-Za-z]`
- **Chiffre:** `[0-9]`
- **Identificateur:** `{lettre}({lettre}|{chiffre})*`

  `ou bien [A-Za-z][A-Za-z0-9]*`
- **Nombre entier signé:** `[-+]?{chiffre}+`

  `ou bien[-+]?[0-9]+`
- **Nombre réel:** `[-+]?{chiffre}+(,{chiffre}+)?`

  `ou bien [-+]? [0-9]+(,[0-9]+)?`

Automate à états finis

- Les langages sont reconnus par des machines formelles appelées: **automates** qui étant donnée un mot sont capable de dire s'il appartient ou pas à un langage.
- Un langage sur un alphabet Σ est régulier si et seulement si il est reconnu par un automate à états finis **[Théorème de Kleene]**. Alors, chaque expression régulière *M*, possède un automate équivalent qui reconnaît *L[M]*.
- **Un automate à états finis (AF)** est un modèle d'un système et de son évolution, c'est-à-dire une description formelle du système et de la manière dont il se comporte.

Automate à états finis

- **Un automate à états finis (AF)** est composé d'un ensemble fini d'états (représentés graphiquement par des cercles), d'une fonction de transition décrivant l'action qui permet de passer d'un état à l'autre, d'un état initial, d'un ou plusieurs états finaux.
- Un AF est donc un graph orienté où les nœuds correspondent aux états et les arcs contiennent les lettres de l'alphabet Σ.

Automate à états finis

- Un automate à états finis **M** est un multiple $(Q, \Sigma, \delta, q_0, F)$ où :
- Σ : est un alphabet ;
- Q : est un ensemble fini d'états ;
- δ : $Q \times \Sigma \rightarrow Q$ est la fonction de transition ;
- q_0 : est l'état initial ;
- F : est un ensemble d'états finaux.

Propriété :

- Le langage **L(M)** reconnu par l'automate M est l'ensemble $\{w \mid \delta(q_0, w) \in F\}$ des mots permettant d'atteindre un état final à partir de l'état initial de l'automate.

Automate à états finis

Exemple:

- Automate à états finis correspondant à l'expression régulière **ab*c(c*| b+c+)** :

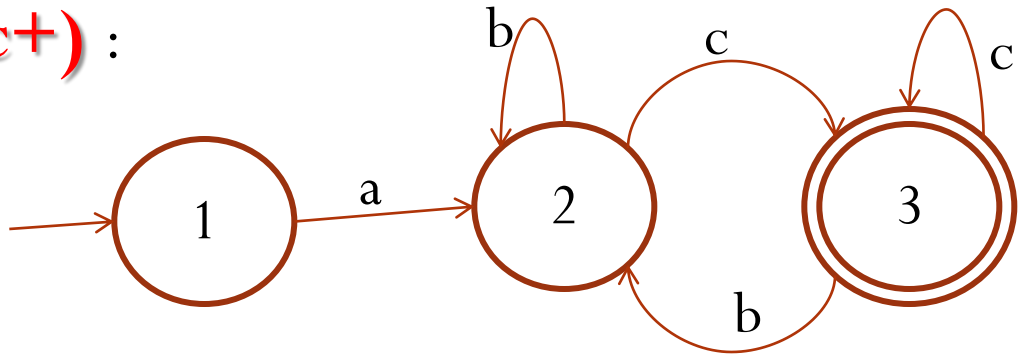


Figure II.1. Automate à états finis pour **ab*c(c*| b+c+)**.

$\Sigma = \{a, b, c\}$

$Q = \{1, 2, 3\}$

$\delta = \{(1, a) \rightarrow 2, (2, b) \rightarrow 2, (2, c) \rightarrow 3, (3, b) \rightarrow 2, (3, c) \rightarrow 3\}$

Etat initial = $\{1\}$

Etats finaux: un seul état final = $\{3\}$.

Représentation d'un automate

- La fonction **δ** de domaine fini **Q×Σ** peut être représentée par une **matrice de dimension 2** dont les éléments sont :
  - **les états** (pour un automate déterministe) ou
  - **un ensemble d'états** (pour un automate non-déterministe)

	a	b	c
→{1}	{2}	-	-
{2}	-	{2}	{3}
#{3}	-	{2}	{3}

Table de transition de l'automate précédent

Automate à états finis déterministe (AFD) et non déterministe (AFND)

AFD

- Un automate à états finis est déterministe si :
  - Pour chaque lettre et chaque état on peut avoir une seule transitions sortante.

ET

  - Pas de transition par ε

AFN

- Un automate à états finis est non déterministe si :
  - Pour un état et une lettre on peut avoir plusieurs transitions sortantes .

OU

  - On peut avoir des transitions par ε

Implémentation d'expression régulières

- Pour effectuer l'analyse lexicale sur les ordinateurs, **les expressions régulières** sont transformées en **automates à états finis** dont l'implémentation est **simple** et la reconnaissance des **unités lexicales** est **rapide**. Ce procédure passe par les 4 étapes suivantes:
- **1er étape:** Transformation des expressions régulières en AFN.
- **2eme étape**: Transformation des AFN en AFD**.**
- **3eme étape**: Minimisation des AFD.
- **4eme étape**: Implémentation des AFD minimaux.

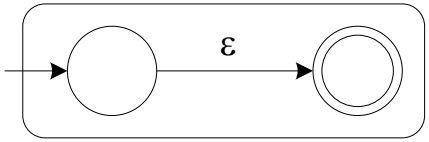
Transformation d'une expression régulière en AFN

Construction de Thompson:

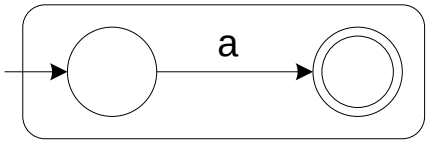
- Parmi les méthodes les plus utilisées pour construirai des automates à états finis à partir des expressions régulières, on trouve la **Construction de Thompson** qui génère automatiquement un **AFN** à partir d'une **ER** comme suit:
- L'**ER** est décomposé en composants simples, et pour chaque composant on construit un automate selon **les règles de Thompson de base**. Ensuite les automates obtenus dans la première étape sont composé pour construire l'automate final selon **les règles de composition de Thompson**.

Règles de Thompson
cas de base

- **1ere règle:** permet de construire un automate pour l'expression régulière **ε.**

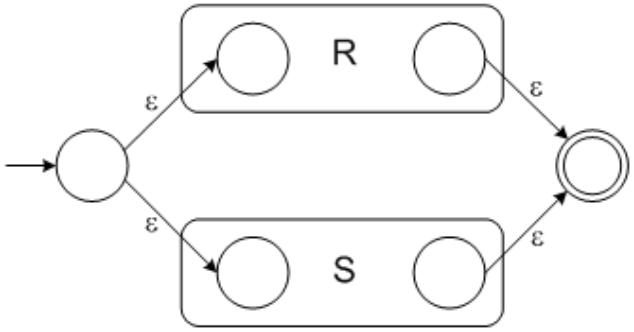


- **2eme règle:** permet de construire un automate pour l'expression régulière **a**.

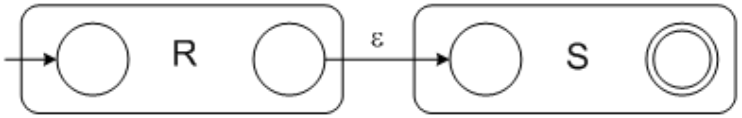


Règles de Thompson
cas de composition

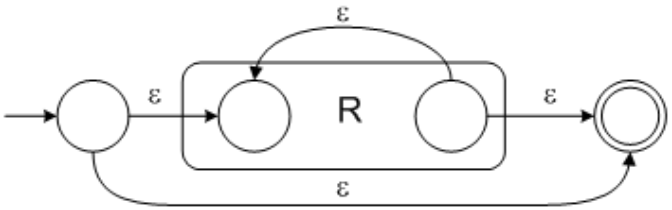
- **3eme règle: Alternation R|S :**



- **4eme règle: Concaténation RS :**



- **5eme règle: Etoile R* :**



Règles de Thompson
Exemple

- L'AFN obtenu a partir de l'ER: **a(b|c)*.**

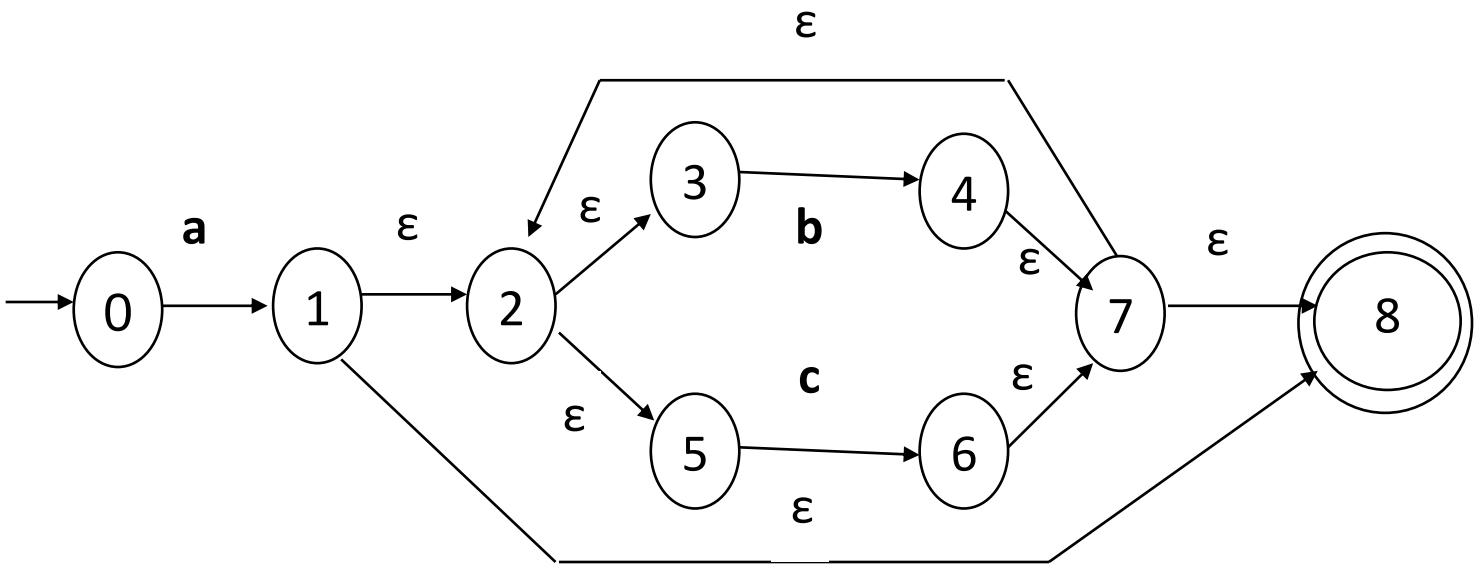


Figure II.2. AFN pour l'ER: **a(b|c)*.**

Transformation d'un AFN en AFD
Algorithme de transformation

Algorithme de transformation

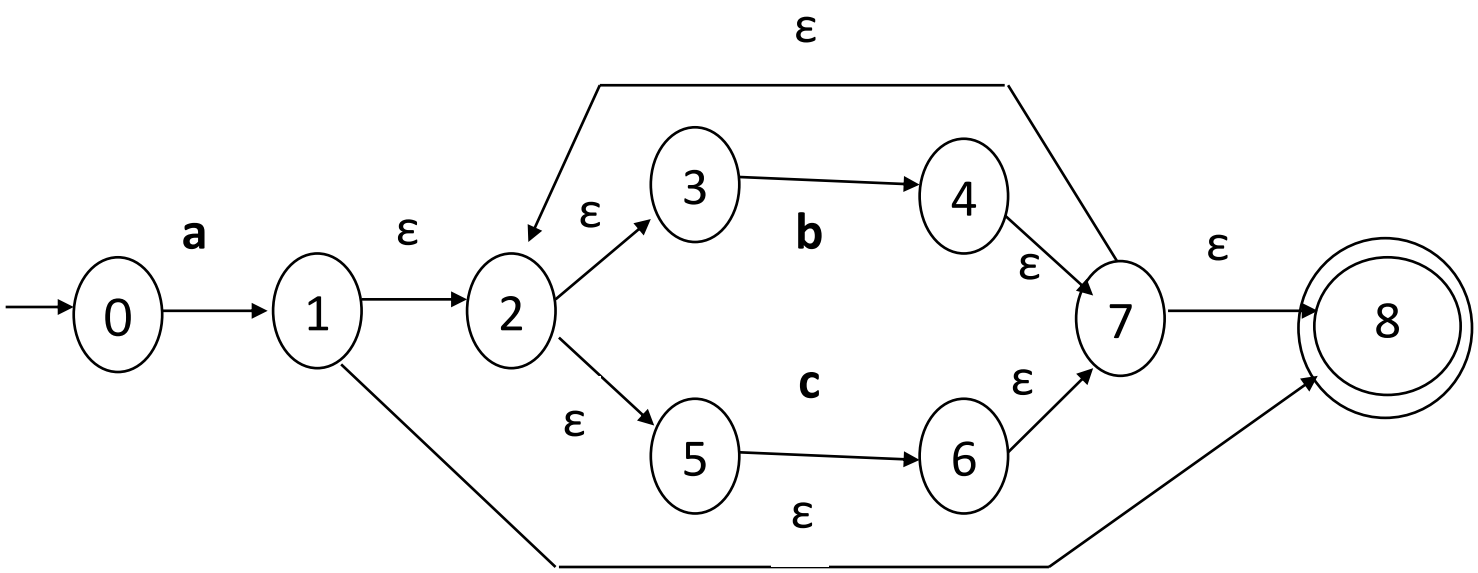
- **Données** : Un **AFN** définissant le langage que **N**.
- **Résultat** : Un **AFD** définissant le même langage que **N**.
- **D** est la table de transition de **AFD**.
- On dispose des fonctions suivantes :
- **ε-fermeture(e) :** ensemble des états de l'AFN accessibles depuis l'état **e** de l'AFN par **ε-transitions** (état **e** inclus).
- **ε-fermeture(T) :** ensemble des états de l'AFN accessibles depuis un état **e** appartenant à **T** par des **ε-transitions** (l'ensemble **T** inclus).
- **Transiter(T, a) :** ensemble des états de l'AFN vers lesquels il existe une transition dans l'AFN sur le symbole **a** à partir d'un état **e** appartenant à **T**.

Transformation d'un AFN en AFD
Algorithme de transformation

E0 = ε-fermeture(Etat initial(AFN));
ajouter **E0** comme **état initial** de **D** (sans le marquer);
tant que un état **E** de **D** est non marque **faire**
    marquer **E**;
    pour chaque caractère **c** de l'alphabet **faire**
        F = ε-fermeture (transiter(E, c));
        si **F** n'est pas un état de **D alors**
            ajouter l'état **F** a **D** (sans le marquer);
            si un élément de **F** est un état d'acceptation de **AFN alors**
                F est un état d'acceptation de **D**
            fin si
        fin si
        ajouter la transition **E** $\xrightarrow{c}$ **F à D;**
    fin pour
fin tant que
fin

Transformation d'un AFN en AFD Exemple

- Prenons l'exemple de **AFN** de la Figure II.2 correspondant à l'expression régulière **a.(b|c)*** :



Transformation d'un AFN en AFD Exemple

Construction de **l'AFD** :

- ε-fermeture(0) = {0}
- Table de transition **D** de **l'AFD** :

	a	b	c
→ **A ={0}**	{1,2,3,5,8}=**B**	-	-
B#={1,2,3,,5,8}	-	{4,7,8,2,3,5}=**C**	{6,7,8,2,3,5}=**D**
C#={4,7,8,2,3,5}	-	**C**	**D**
D#={6,7,8,2,3,5}		**C**	**D**

- Etat initial : ε-fermeture(0) = {0}=**A**
- Etats finaux : B, C , D

Transformation d'un AFN en AFD
Exemple

- **l'AFD** obtenu pour l'ER: **a.(b|c)*** en utilisant l'algorithme de transformation.

	a	b	c
→ A	B	-	-
B#	-	C	D
C#	-	C	D
D#		C	D

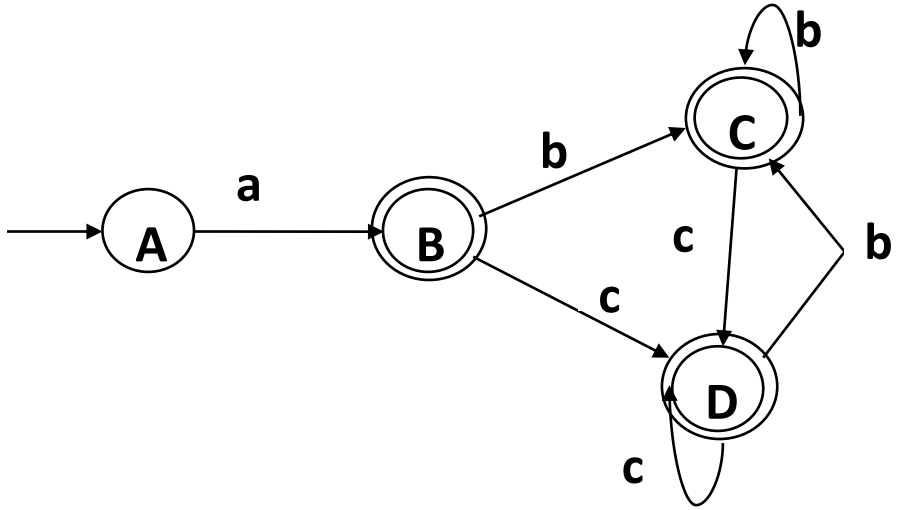


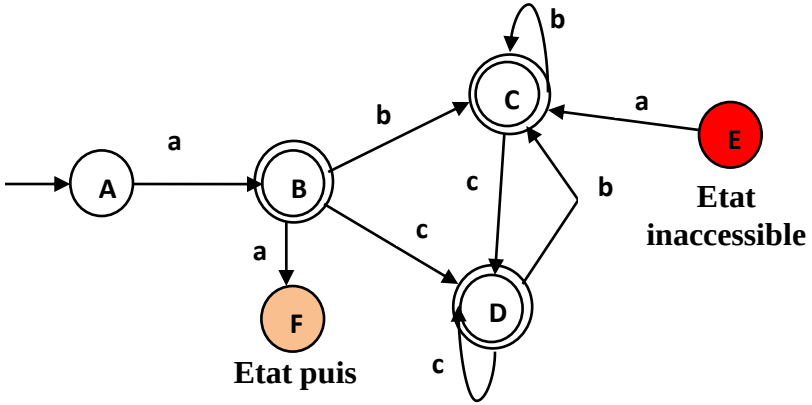
Figure II.4. AFD pour l'ER: **a.(b|c)***

Minimisation de l'AFD

- Pour effectuer l'analyse lexicale, il est préférable que la table d'analyse de **L'AFD** soit la plus petite possible afin d'économiser la mémoire.
- **Théorème:** Il existe un automate déterministe unique avec un nombre **minimal** d'états reconnaissant un langage rationnel **L [Myhill-Nerode]**
- L'algorithmes de raffinements de partitions (ex: l'algorithme de **Moore et** l'algorithme de **Hopcroft**) sont les plus simple à utiliser.

Minimisation de l'AFD

- Avant l'utilisation de l'algorithmes de raffinements de partitions. Il faut supprimer les états **inaccessibles** et les états **puis**.
- les états **inaccessibles** sont des états ou on ne peut pas attendre cet état à partir d'un état initial.
- les états **puis** sont les états ou il n'a pas un chemin vers un état final à partir de cet état.



Minimisation de l'AFD
Algorithme

Soit **A= $(Q, \Sigma, \delta, q_0, F)$** un automate fini déterministe

- Initailement: Créer deux Classes d'états **C1** et **C2** // **C1** contient les **états finaux (F)** et **C2** contient les **états non-finaux (Q\F)**

Répéter

  Pour chaque partition **Ci** **faire**

    Pour chaque symbole d'entré **a** **faire**

      Si il existe deux états différents **q1** et **q2** appartient à **Ci** qui lisent le symbole **a** et mènent vers des états appartient aux deux classes différentes **alors**

        Créer une nouvelle classe **Cj** et séparer **q1** de **q2**.

      Fin si

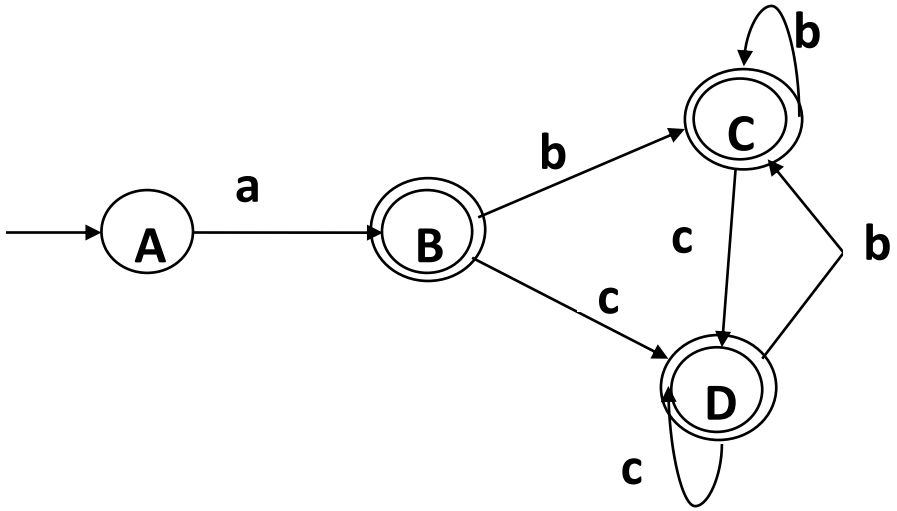
    Fin

  Fin pour

Jusqu'à ce que il n y a pas de deux classes à séparer.

Algorithme de minimisation
Exemple

- Prenons l'exemple de **AFN** de la Figure II.4 correspondant à l'expression régulière **a.(b|c)***



Algorithme de minimisation
Exemple

- Prenons l'exemple de **AFD** de la **Figure II.4** correspondant à l'expression régulière **a.(b|c)*.**
- **Initialement**:

  I: C1: {A} , C2:{ B,C,D}

  B---b--> C C---b--> C D---c--> D

  B---c--> D C---c--> D D---c--> D

- **1ière Itération:**

  Π : C1: {A} , C2:{ B,C,D}

- On ne peut scinder {B,C,D} puisque B, C et D sont des états inséparables.

Algorithme de minimisation Exemple

- **Table de transition de l'AFD minimal.**

	a	b	c
→A	B		-
B#	-	B	B

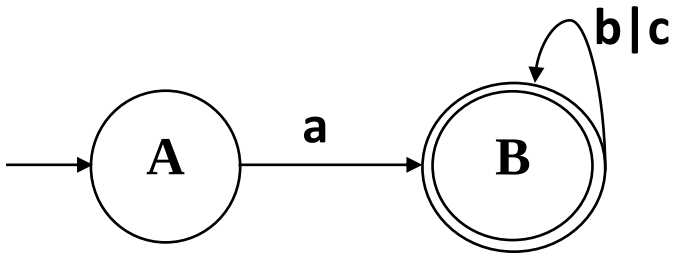
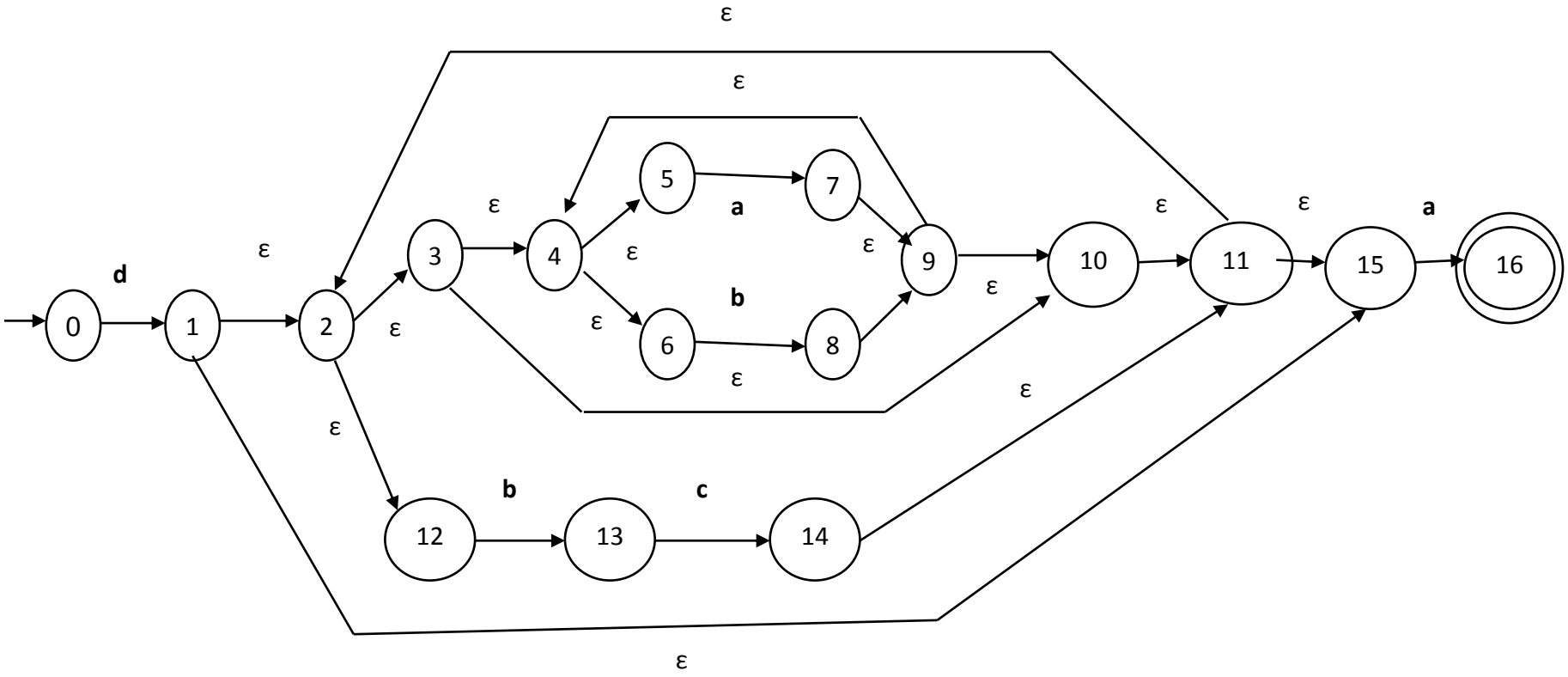


Figure II.5. AFD minimal pour l'ER: **a.(b|c)***

Expression régulières et automate Exemple 2

- L'AFN obtenu a partir de l'ER: **d((a|b)*|bc)*a.**
- **ER →AFN**

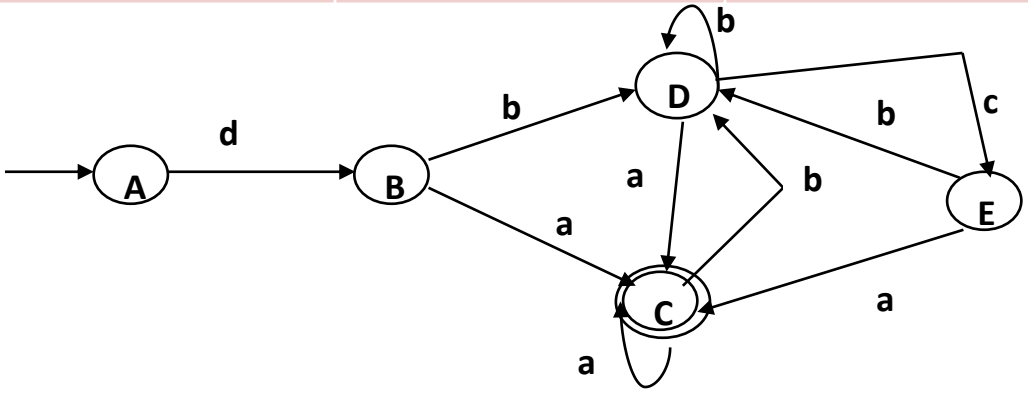


Expression régulières et automate Exemple 2

- **AFN →AFD**

	a	b	c	d
→A = {0}	-	-	-	{1,2,3,4,5,6,10,11,12,15}=**B**
B	{7,9,10,11,15,4,5,6,2,3,12,16}=**C**	{8,13,9,10,11,15,4,5,6,2,3,12}=**D**	-	-
C#	**C**	**D**	-	-
D	**C**	**D**	{14,11,15,2,3,4,5,6,10,12}=**E**	-
E	**C**	**D**	-	-

Figure II.6. AFD pour l'ER: **d((a|b)*|bc)*a**



Expression régulières et automate Exemple 2

- **AFD →AFD Minimal**
- **Initialement**: I: {A,B,E,D} , {C}
- **1ière Itération:** II : {A},{B,D,E}, {C}

  $A \xrightarrow{a} \Phi$

  $B \xrightarrow{a} C$ $B \xrightarrow{b} D$ $B \xrightarrow{c} \Phi$ $B \xrightarrow{d} \Phi$

  $E \xrightarrow{a} C$ $E \xrightarrow{b} D$ $E \xrightarrow{c} \Phi$ $E \xrightarrow{d} \Phi$

  $D \xrightarrow{a} C$ $D \xrightarrow{b} D$ $D \xrightarrow{c} E$
- **2ième Itération**: III: {A},{D},{B,E}, {C}
- II = III. On ne peut scinder {B,E} puisque B et E mènent vers les mêmes états avec tous les symboles d'entrés.

Expression régulières et automate Exemple 2

	a	b	c	d
→A	-	-	-	BE
BE	C	D		
D	C	D	BE	-
C#	C	D	-	-

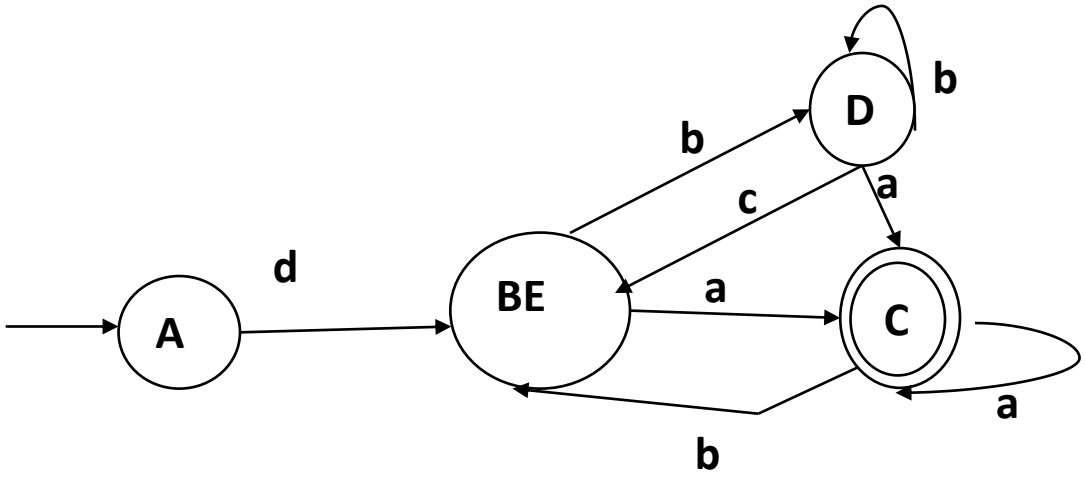


Figure II.7. AFD minimal pour l'ER: **d((a|b)*|bc)*a**

Implémentation des AFD minimaux Algorithme de reconnaissance

- **Entrée**: une chaine de caractère d'entrée **S** terminée par un caractère spécial "#".
- **Sortie**: Chaine reconnue par l'automate ou non.
- On dispose des fonctions suivantes :

✓ **Transiter(e, c) :** donne l'état de l'automate vers lequel il existe une transition depuis l'état **e** sur le caractère d'entrée **c**.

✓ **CarSuiv ( ) :** retourne le prochain caractère à analyser de la chaine **S**.

Implémentation des AFD minimaux
Algorithme de reconnaissance

```
e:=e0;
c:=CarSuiv();
Tant que (c ≠ '#' et e ≠ ∅)
Faire
        e:=Transiter(e,c);
        c:=CarSuiv();
Fait;
Si e∈F
Alors "Chaine acceptée"
Sinon "Chaine refusée"
Fsi
```

Outils pour Implémenter les ER

- Implémentions dure des automates d'états finis.
- Utilisation des librairies existantes telles que : Java, C++, PHP, … etc
- Utilisation de générateurs des analyseurs lexicaux: Lex, Flex, Jlex, … etc.

LEX: générateur d'analyseurs lexicaux Introduction

- **Lex** est un outil de génération d'analyseurs lexicaux en langage **C**. Il a été originellement écrit par **Mike Lesk** et **Eric Schmidt** en 1975.
- **Lex** est capable de traiter des langages de type 3 (réguliers).
- **Lex** est fréquemment utilisé en association avec le générateur d'analyseur syntaxique **Yacc**.

LEX: générateur d'analyseurs lexicaux Principe

- **Lex** prend en entrée la définition des **unités lexicales** sous forme des **expressions régulières**.
- **Lex** produit un automate fini déterministe minimal permettant de reconnaître les unités lexicales.
- **Lex** produit l'automate sous la forme d'un programme **C**.

LEX: générateur d'analyseurs lexicaux
Expressions régulières en LEX (1)

Symbole	Signification
x	Le caractère '**x**'
.	N'importe quel caractère sauf **n**
[xyz]	Soit **x**, soit **y**, soit **z**
[^bz]	Tous les caractères, sauf **b** et **z**
[a-z]	N'importe quel caractère entre **a** et **z**
[^a-z]	Tous les caractères, sauf ceux compris entre **a** et **z**
R*	Zéro **R** ou plus, ou **R** est n'importe quelle expression régulière
R+	Un **R** ou plus
R?	Zéro ou un **R** (c'est-a-dire un **R** optionnel)
R{2,5}	Entre deux et cinq **R**

LEX: générateur d'analyseurs lexicaux
Expressions régulières en LEX (2)

Symbole	Signification
R{2,}	Deux **R** ou plus
R{2}	Exactement deux **R**
"[xyz\"foo"	La chaine **'[xyz"foo'**
{NOTION}	L'expansion de la notion **NOTION** definie plus haut
RS	**R** suivi de **S**
R|S	**R** ou **S**
R/S	**R**, seulement s'il est suivi par **S**
^R	**R**, mais seulement en début de ligne
R$	**R**, mais seulement en fin de ligne
<<EOF>>	Fin de fichier

LEX: générateur d'analyseurs lexicaux Expressions régulières Exemples (1)

- **blancs** **[\t\n]+**
- **lettre** **[A-Za-z]**
- **chiffre10** **[0-9]** /* base 10 */
- **chiffre16** **[0-9A-Fa-f]** /* base 16 */
- **identificateur** **{lettre}(_|{lettre}|{chiffre10})***

  ou bien **identificateur** **[A-Za-z] [_A-Za-z0-9] ***

LEX: générateur d'analyseurs lexicaux Expressions régulières Exemples (2)

- **chiffre** **[0-9]**
- **entier** **{chiffre}+**
- **exposant** **[eE][+-]?{entier}**
- **reelVF** **{entier}("."{entier})?{exposant}?**
- **reel** **[+-]? [0-9] +("." [0-9] +)?**

LEX: générateur d'analyseurs lexicaux
Structure d'un programme LEX

- Un fichier de description pour **Lex** est formé de trois parties:
  - **Déclarations**
  - **%%**
  - **Productions**
  - **%%**
  - **Code additionnel**
- Aucune partie n'est obligatoire.
- le symbole **%%** est utiliser comme un séparateur entre les parties.

LEX: générateur d'analyseurs lexicaux
Structure d'un pgm LEX (1ere partie)

- **1ere partie**: **Déclarations :** peut contenir :
- Du code écrit dans le langage cible (**C**), encadré par **%{** et **%}**. **Lex** recopie tel quel tout ce qui est écrit entre ces deux signes.
- **Des expressions régulières** définissant des notions non terminales, à utiliser dans le reste de la première partie du fichier **lex**, ainsi que dans la seconde partie, en les mettant entre {et}. Ces spécifications sont de la forme :

notion expression régulière

LEX: générateur d'analyseurs lexicaux
Structure d'un pgm LEX (1ere partie)

- **Exemple :**

```
%{
#include "calc.h"
#include <stdio.h>
#include <stdlib.h>
%}
/* Expressions régulières */
Blancs    [\t\n ]+
Lettre    [A-Za-z]
chiffre   [0-9]
identificateur    {lettre}(_|{lettre}|{chiffre10})*
```

LEX: générateur d'analyseurs lexicaux Structure d'un pgm LEX (2ere partie)

- **2eme partie**: **les productions:**
- Cette partie sert à indiquer à **Lex** ce qu'il devra faire lorsqu'il rencontrera telle ou telle **unité lexicale**. Elle peut contenir des **productions** de la forme :

expression régulière **action**

les **actions** seront écrites dans le code du langage cible **(C)** et doivent être placées entre {et}.

Si **action** est **absente**, **Lex** recopiera les caractères tels quels sur la sortie standard.

LEX: générateur d'analyseurs lexicaux Structure d'un pgm LEX (2ere partie)

- Les commentaires tels que **/* ... */** peuvent être placés **uniquement** dans les **actions** entre parenthèses. Dans le cas contraire, ceux-ci seraient considérés par **Lex** comme **des expressions régulières** ou **des actions**, ce qui donnerait lieu à des **messages d'erreur**.
- La variable **yytext** désigne dans les **actions** les caractères acceptés par une **expression régulière**. Il s'agit d'un tableau de caractère de longueur **yyleng** (donc défini comme **char yytext[yyleng]**).

LEX: générateur d'analyseurs lexicaux
Structure d'un pgm LEX (2ere partie)

- **Exemple** :

```
%%

[ \t]+$      ;

[ \t]        printf(" ");
```

- Ce programme supprime tous les espaces inutiles dans un fichier.

LEX: générateur d'analyseurs lexicaux
Structure d'un pgm LEX (3ere partie)

- **3eme partie**: **Le code additionnel:**
- Tu peux mettre dans cette partie **facultative** tous le code que tu veux. Si tu ne mets rien, **Lex** considère que c'est juste :
- **main() {**

yylex();

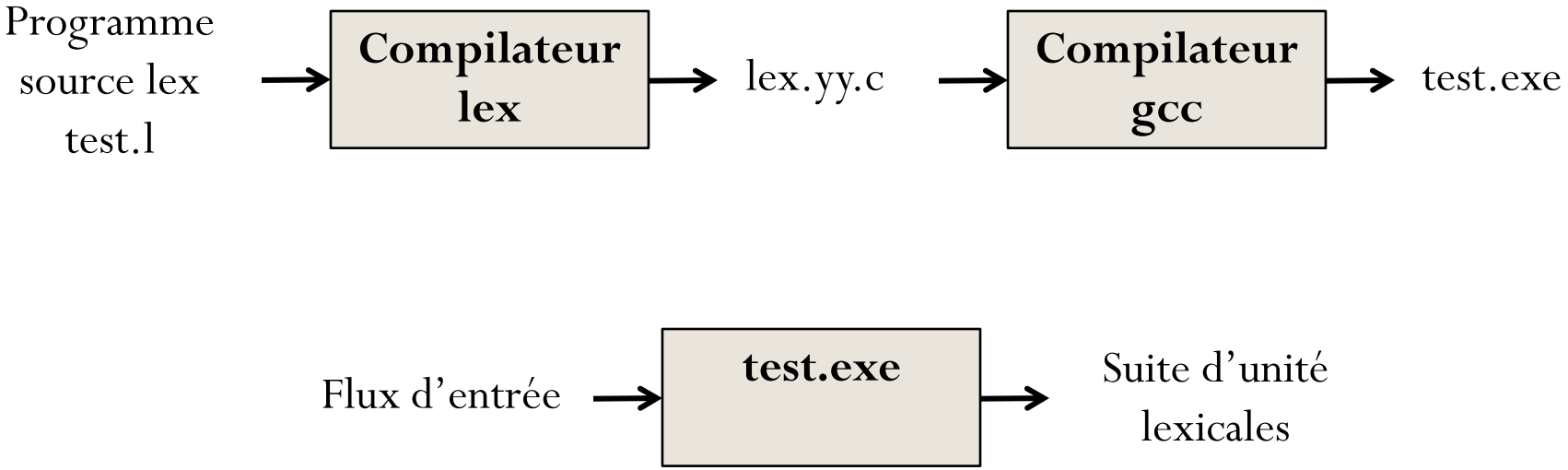
}.

LEX: générateur d'analyseurs lexicaux
Exécution de LEX sous Linux

- le programme **LEX** s'exécute comme suit:
  - Le fichier nommé par exemple **test.l** qui contient la spécification de l'analyseur lexical à produire est compiler avec le compilateur **lex** en exécutant la commande **lex**.
  - La commande **lex** génère le code **C** de l'analyseur qui est mis dans un fichier **lex.yy.c**
  - Le compilateur **gcc** est utilisé pour compiler le fichier **lex.yy.c** et générer un code exécutable (**a.out** par exemple).
  - Le code exécutable est chargé et exécuté.

LEX: générateur d'analyseurs lexicaux
Exécution de LEX sous Linux

- **lex test.l**
- **gcc lex.yy.c -o test.exe -lfl**
- **./test.exe**



LEX: générateur d'analyseurs lexicaux Exemple

```
/* just like Unix wc */
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+   { words++; chars += yyleng; }
\n          { chars++; lines++; }
.           { chars++; }
%%
main(int argc, char **argv)
{
 yylex();
 printf("lines=%d\n words=%d\n words=%d", lines, words, chars);
}
```