الكلية متعددة التخصصات - ورزازات
ⵜⴰⵖⵍⵍⴰⵜ ⵜⴰⵅⵜⵚⵚⵉⵚⵜ – ⵍⵓⵕⵣⴰⵣⴰⵜ
FACULTÉ POLYDISCIPLINAIRE DE OUARZAZATE

Compilation

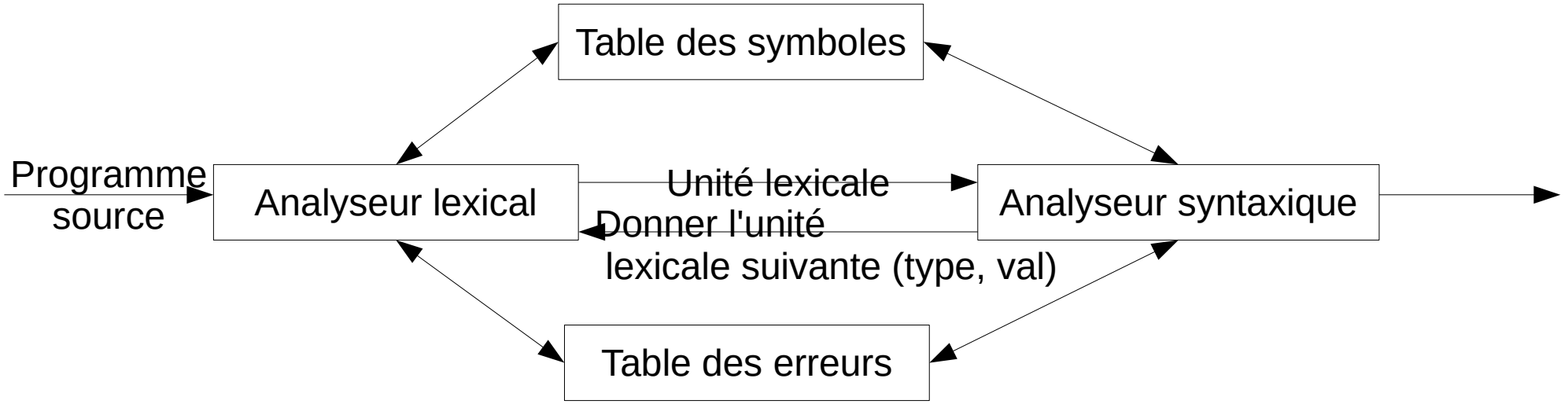
Chapitre2 : Analyse lexicale

SMI-S5
Compilation
Prof. : M. BENADDY

Introduction (1)

- **Rôle de l'analyseur lexicale (scanner) :** lire les caractères d'entrée (programme source) et produire comme résultat une suite d'unités lexicales appelées TOKENS que l'analyseur syntaxique va utiliser.
- Les TOKENS sont constitués au fur et à mesure de la lecture du texte source, on distingue deux types d'unités lexicales.
  - Les unités propres au langage, les mots clés, les séparateurs, les opérateurs, ...etc.
  - Les unités personnalisées qui sont crées par le programmeur (les constantes, les identificateurs...).

Introduction (2)



- **Remarque** : l'analyseur lexical de certains compilateurs procèdent à l'élimination dans le programme source des commentaires et des espaces qu'apparaissent sous forme de caractères blancs ou de tabulations ou bien les sauts de lignes.

Unités lexicales, lexèmes, modèles (1)

- **Token : unité lexicale**, est une paire de nom et d'une valeur optionnelle. Le nom du token est un symbole abstrait qui représente une unité lexicale, (mot clé, suite de caractère dénotant un identifiant, ...).
- **Le modèle** : est une description de la forme que peut prendre les lexèmes d'une unité lexicale.
  - C'est une règle qui décrit l'ensemble des lexèmes pouvant représenter une unité lexicale particulière dans le programme source.
  - Dans le cas des mots clés, le modèle est juste une séquence de caractères qui forme le mot clé.
  - Pour les identificateurs et d'autres unités lexicales, le modèle est une structure plus complexes.
- **Lexème** : est une suite de caractères dans le programme sources qui concorde avec le modèle.
  - Il est identifié par l'analyseur lexical comme instance d'un token.

Unités lexicales, lexèmes, modèles (2)

- **Exemple :**

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Unités lexicales, lexèmes, modèles (3)

- **Remarques :** Dans la plupart des langages de programmation, les classes suivantes couvrent la majorité des tokens (unités lexicales).
  - Un token (unité lexicale) pour chaque mot clé. Le modèle pour le mot clé est le même que le mot clé.
  - Les tokens pour les opérateurs, soit individuels soit par classes (exemple comparaison de l'exemple précédent).
  - Une unité lexicale pour tous les identificateurs (nom des variables, des fonctions, des classes, tableaux, structures...).
  - Une ou plusieurs unités lexicales (tokens) pour les constantes, les nombres et les chaînes de caractères.
  - Une unité lexicale pour chaque symbole de ponctuation, tels que, la virgule, le point virgule, les parenthèses, les accolades,...

Unités lexicales, lexèmes, modèles (3)

- **Exemple :** float pi = 3.14 ;
  - La sous chaîne pi est un lexème d'unité lexicale identificateur, cette unité lexicale est retournée à l'analyseur syntaxique, le retour est souvent réalisé en passant un entier correspondant à l'unité lexicale. Le modèle de l'unité lexicale float est l'unité correspondant à la simple chaîne float qui vérifie l'orthographe des mots clés.

Attributs des unités lexicales (1)

- L’analyseur lexical réduit les informations sur les unités lexicales dans les attributs qui leur sont associés.
- En pratique une unité lexicale a en général un seul attribut : un pointeur vers l’entrée dans la table des symboles dans laquelle l’information sur l’unité lexicale est conservée, le pointeur devient alors l’attribut de l’entité lexicale.

Attributs des unités lexicales (2)

- **Exemple** : en Fortran E = M * C * * 2 (**désigne la puissance en Fortran)
- Les unités lexicales et les valeurs des attributs sont donnés par :

<id, pointeur vers l'entrée de la table des symboles associé à E>

<op_affectation>

<id, pointeur vers l'entrée de la table des symboles associé à M>

<op_multiplication>

<id, pointeur vers l'entrée de la table des symboles associé à C>

<op_exposant>

<nombre, valeur entière 2>

Les mots réservés

- Certains langages ne tolèrent pas que les mots clés soient utilisés comme des identificateurs.
- Ce qui veut dire que leur signification est prédéfinie et ne peut pas être changé par le programmeur.
- Les mots clés sont installés dans la table des symboles et quand un identificateur est rencontré il est comparé avec les mots clés de la table de symboles, s'il coïncide avec un des mots clés de la table le scanner (analyseur lexicale) génère le code de ce mot clés, sinon le code de l'identificateur est transmis.
- **Exemples** : Pascal et C

Les erreurs lexicales

- Peu d'erreurs sont détectées au niveau lexicale, car un analyseur lexical a une vision très localisée du programme source.
- **Exemple** : si on rencontre la chaîne fi dans un programme avec fi(i==j) l'analyseur lexical ne peut pas dire s'il s'agit du mot clé if mal écrit ou d'un identificateur non déclaré.
- **Exemple** : une variable qui commence par un caractère non valide (μx, αy, ...) à ce moment-là l'analyseur lexical déclenche une erreur qui indique la non validité du nom de la variable.

Spécification des unités lexicales (1)

- La procédure d'un analyseur lexical consiste dans un premier temps en la description des unités lexicales à l'aide des ***expressions régulières***.
- Les expressions régulières sont une notation importante pour spécifier les modèles, chaque modèle reconnaît un ensemble de chaînes appelés aussi mots, ces expressions régulières seront transformées en des ***automates*** qui seront implémentés sur machine.

Spécification des unités lexicales (2)

- **Mots et langage :**
- **Lettre et mots :**
  - Soit A un ensemble fini appelé alphabet, les éléments de A sont appelés des lettres, exemple : A={a,b,c,1,3,h}.
  - Un mot ou une chaîne sur un alphabet est une séquence finie de symboles extrait de cet ensemble.
  - On définit A^+ l'ensemble des mots non vides comme le plus petit ensemble tel que :
    - Pour toute lettre ℓ appartenant à l'alphabet A, ℓ appartiendra à A^+.
    - Pour tout mot $m \in A^+$ et toute lettre $\ell \in A$, le mot $m\,\ell \in A^+$, ($m\,\ell$ représente la concaténation du mot m et de la lettre ℓ).
    - Soit un mot $m \in A^+$, on dira que m constitue un mot sur A (mot fait à partir de l'alphabet A).

Spécification des unités lexicales (3)

- **Mots et langage :**
- **Opérations sur les langages :**
  - Plusieurs opérations peuvent s'appliquer aux langages.
  - Pour l'analyse lexicale on s'intéresse principalement à l'union, la concaténation et la fermeture (de Kleene), on peut aussi généraliser l'opération d'exponentiation aux langages en définissant $L^0 = \{ \varepsilon \}$, $L^i = LL^{i-1}$ ainsi L^i est la concaténation de L, i-1 fois avec lui même.

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

  - **Remarque** : Ces notations sont employées dans l'écriture des expressions régulières.

Spécification des unités lexicales (4)

- **Mots et langage :**
- **Opérations sur les langages :**
- **Exemple :**
- L = {a, ...,z,A,...,Z}, C={0,...,9}, on peut créer de nouveaux langages à partir des langages L et C, en appliquant les opérateurs précédents.
- L U C est l'ensemble des lettres et des chiffres.
- LC est l'ensemble des mots formés d'une lettre suivie d'un chiffre.
- L^4 est l'ensemble des mots formés de 4 lettres.
- L* est l'ensemble de tout les mots de lettres, y compris le mot vide, Ɛ.
- L(LUC)* est l'ensemble des mots de lettres et de chiffres commençant par une lettre.
- C^+ est l'ensemble des mots qui sont constitués d'au moins d'un chiffre.

Spécification des unités lexicales (5)

- **Expressions régulières :**
- On construit une expression régulière à partir d'expressions régulières plus simples, en utilisant les règles de définition du langage concerné.
- Chaque expression régulière *r* dénote un langage L(*r*). Les règles de définition spécifient comment L(*r*) est formé en combinant de manière variée les langages dénotés par les sous-expressions de *r*.
- Les expressions régulières sont construites récursivement à partir d'expressions régulières plus simples, en utilisant des règles.
- Chaque expression régulière *r* dénote un langage L(r), qui est aussi définit récursivement à partir des langages dénotés par les sous-expressions de *r*.
- Les règles qui définissent les expressions régulières sur un alphabet ∑ et les langages dénotés par ces expressions.

Spécification des unités lexicales (6)

- **Expressions régulières :**
- **Règles :**
  - Ɛ est une expression régulière, et L(Ɛ) = {Ɛ}, le langage dont le seule membre est le mot vide Ɛ.
  - Si a est un symbole de ∑, alors a est une expression régulière, et L(a) = {a}, c-à-d le langage avec une seule chaîne de caractère, de longueur 1.

Spécification des unités lexicales (7)

- **Expressions régulières :**
- **Induction :**
  - On suppose que *r* et *s* sont deux expressions régulières dénotant les deux langages L(r) et L(s) respectivement.
  - Il existe quatre règles d'induction qui permettent de construire des expressions complexes à partir d'expressions régulières plus simples.
    - (r)|(s) est l'expression régulière qui dénote le langage L(r)UL(s)
    - (r)(s) est l'expression régulière qui dénote le langage L(r)L(s),
    - (r)* est l'expression régulière dénotant le langage (L(r))*,
    - (r) est l'expression régulière qui dénote L(r). Cette règle signifie qu'on peut ajouter des parenthèses sans changer le langage qu'elle dénote.

Spécification des unités lexicales (8)

- **Expressions régulières :**
- **Priorité des opérateurs :**
  - L'opérateur * a la plus haute priorité et est associatif à gauche.
  - La concaténation a la deuxième priorité et est associatif à gauche.
  - Le ou (|) a la faible priorité.
  - En considérant ces conventions, par exemple l'expression régulière (a)|((b)*(c)) est équivalente à a|b*c, les deux dénotent l'ensemble des mots formés d'un seule ou zéro a ou bien zéro ou plusieurs b suivi d'un c.

Spécification des unités lexicales (9)

- **Expressions régulières :**
- **Exemple :** soit l'alphabet ∑ ={a,b}
  - L'expression régulière a|b dénote le langage {a,b}
  - (a|b)(a|b) dénote {aa,ab,ba,bb}, le langage de mots de longueur 2 sur l'alphabet ∑, une autre expression régulière est aa|ab|ba|bb.
  - a* dénote le langage des mots formés de zéro ou plus de a, {Ɛ, a, aa, aaa, ...}.
  - (a|b)* dénote l'ensemble des mots formés de zéro ou plusieurs a ou b, {Ɛ, a, b, aa, ab, ba, bb, aaa, ...}. (a|b)* = (a*b*)*=(a*|b*)*.
  - a|a*b dénote le langage {a, b, ab, aab, aaab, ...}, c-à-d le mot a et l'ensemble des mots formés de zéro ou plusieurs a qui finissent par b.

Spécification des unités lexicales (10)

- **Expressions régulières :**
- **Remarque** : Un langage qui peut être définit par une expression régulière est appelé un ensemble régulier. Si deux expressions régulières *r* et *s* dénotent le même ensemble régulier, on dit qu'ils sont équivalents *r*=*s.*

Spécification des unités lexicales (11)

- **Expressions régulières :**
- **Propriétés algébriques des expressions régulières :**

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Spécification des unités lexicales (12)

- **Expressions régulières :**
- **Définitions régulières :** Une définition régulière est une suite de définitions de la forme :

$$\begin{array}{lcl} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \cdots & \\ d_n & \rightarrow & r_n \end{array}$$

où

- Chaque d_i est un nouveau symbole distinct qui n'appartient pas à Σ
- Chaque r_i est une expression régulière sur les symboles : Σ U $\{d_1,d_2,\ldots,d_{i-1}\}$

Spécification des unités lexicales (13)

- **Expressions régulières :**
- **Définitions régulières :**
- **Exemple 1** : les identificateurs en C sont des mots formés de lettres, chiffres, et underscore (_). Une définition régulière de cet ensemble est la suivante :

lettre → A|B|...|Z|a|b|...|z| _
chiffre → 0|1|...|9
id → lettre(lettre|chiffre)*

Spécification des unités lexicales (14)

- **Expressions régulières :**
- **Définitions régulières :**
- **Exemple 2** : Les nombres non signés (entiers ou réels) sont des chaînes, comme 5280, 0.01234, 6.336E4, ou 1.89E-4. La définition régulière de cette catégorie est donnée par l'expression suivante :

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\
\mathit{digits} & \rightarrow & \mathit{digit}\ \mathit{digit}^{*} \\
\mathit{optionalFraction} & \rightarrow & \mathtt{.}\ \mathit{digits} \mid \epsilon \\
\mathit{optionalExponent} & \rightarrow & (\mathtt{E} (\mathtt{+} \mid \mathtt{-} \mid \epsilon)\ \mathit{digits}) \mid \epsilon \\
\mathit{number} & \rightarrow & \mathit{digits}\ \mathit{optionalFraction}\ \mathit{optionalExponent}
\end{array}
$$

Spécification des unités lexicales (15)

- **Expressions régulières :**
- **Extensions aux expressions régulières :**
  - Les opérateurs de base (l'union, la concaténation et la fermeture) ont été introduites par Kleene vers 1950, des extensions on été ajoutées par la suite pour pouvoir définir quelques modèles de chaînes.
  - Les extensions suivantes sont incorporées dans utilitaires sous UNIX tel que Lex, qui sont utiles dans l'écriture d'analyseurs lexicaux.

Spécification des unités lexicales (16)

- **Expressions régulières :**
- **Extensions aux expressions régulières :**
  - L'opérateur unaire + représente, la fermeture positive + de l'expression régulière et son langage. Si *r* est une expression régulière, alors $(r)^+$ dénote le langage $(L(r))^+$. L'opérateur + a la même priorité et associativité que l'opérateur * (Kleene closure), on peut écrire alors, $r^* = r^+|\varepsilon$ et $r^+ = rr^* = r^*r$.
  - Zéro ou une instance : L'opérateur unaire ? signifie ''zéro ou une occurrence''. C-à-d *r* ? est équivalente à $r|\varepsilon$, d'une autre façon $L(r?)=L(r)\cup\{\varepsilon\}$. L'opérateur ? a les mêmes priorité et associativité que * et +.
  - Classes de caractères : L'expression régulière $a_1|a_2|...|a_n$, où les a_i sont des symboles de l'alphabet, on peut alors les remplacer par $[a_1a_2...a_n]$. quand $a_1,a_2,...,a_n$ forment une séquence logique, c-à-d des lettres majuscules consécutives, des lettres minuscules, ou des chiffres, on peut les remplacer par a_1-a_n, (on prend juste le premier et dernier élément séparés par un tiret -). Par conséquent [abc] ≡ a|b|c, et [a-z] ≡ a|b|...|z.

Spécification des unités lexicales (17)

- **Expressions régulières :**
- **Extensions aux expressions régulières :**

$r^* = r^+ \mid \varepsilon$ et $r^+ = rr^* = r^*r$	Fermeture
r? = $r \mid \varepsilon$	Zéro ou une instance
[abc] ≡ a|b|c, et [a-z] ≡ a|b|...|z	Un élément de la classe

Spécification des unités lexicales (18)

- **Expressions régulières :**
- **Extensions aux expressions régulières :**
- **Exemple :** en utilisant ces formes abrégées réécrire les expressions régulières suivantes :

lettre → A|B|...|Z|a|b|...|z| _
chiffre → 0|1|...|9
id → lettre(lettre|chiffre)*

Spécification des unités lexicales (19)

- **Expressions régulières :**
- **Extensions aux expressions régulières :**
- **Exemple :** en utilisant ces formes abrégées réécrire les expressions régulières suivantes :

lettre → [A-Za-z_]
chiffre → [0-9]
id → lettre(lettre|chiffre)*

Reconnaissance de unités lexicales (1)

- Comment utiliser les modèles des unités lexicales, pour construire un programme qui examinera le texte d'entrée pour décider si les lexèmes se sont conforme avec les modèles. Considérons cet exemple :

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

Reconnaissance de unités lexicales (2)

- Les unités lexicales concernées par l'analyseur lexical sont **if**, **then**, **else**, **relop**, **id**, et **number**, ils sont aussi les terminaux de la grammaire pour les instructions de branchement conditionnel.
- Les définitions régulières définissant les modèles de ces unités lexicales sont :

digit	→	`[0-9]`
digits	→	*digit*$^+$
number	→	*digits* (`.` *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

Reconnaissance de unités lexicales (3)

- Pour ignorer les commentaires et les blancs l'analyseur lexical doit les reconnaître pour cela, on définit l'unité lexicale ''ws'' comme suite :

$$ws \rightarrow (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^{+}$$

- Les caractères blancs (blank), tabulation (tab) et nouvelle ligne (newline) sont des symboles abstraits utilisés pour exprimer les caractères ASCII désignant la même chose.
- L'unité lexicale *ws* est reconnue par l'analyseur lexical, mais elle ne sera pas retournée à l'analyseur syntaxique, mais, on reprend l'analyse qui suit l'espace blanc.

Reconnaissance de unités lexicales (4)

- Le but de l'analyseur lexicale est résumé dans le tableau ci-dessous qui montre pour chaque lexème ou famille de lexèmes, qu'elle unité lexicale (token) qui sera retournée à l'analyseur syntaxique ainsi que sa valeur d'attribut.

Lexèmes	Nom de l'unité lexicale	Valeur de l'attribut
N'importe quel *ws*	-	-
if	**if**	-
then	**then**	-
else	**else**	-
N'importe quel *id*	**id**	**Pointeur vers la table des symboles**
N'importe quel *number*	**number**	**Pointeur vers la table des symboles**
<	**relop**	**LT**
<=	**relop**	**LE**
=	**relop**	**EQ**
<>	**relop**	**NE**
>	**relop**	**GT**
>=	**relop**	**GE**

Les automates à états finis (1)

- Un reconnaisseur pour un langage est un programme qui prend en entrée une chaîne *x* et répond oui si *x* est une phrase du langage et non autrement.
- On compile une expression régulière en un reconnaisseur en construisant un diagramme de transition appelé automate fini.
- Un automate fini est capable de reconnaître précisément les ensembles réguliers.
- Il existe types d'automates à états finis:
  - **Les automates à états finis non déterministes** (**AFN**) n'ont aucune restrictions sur les étiquettes de leurs arcs. Un symbole peut étiqueter plusieurs arcs partant d'un même état, et la chaîne vide ε est une étiquette possible.
  - **Les automates à états finis déterministes** (**AFD**), pour lesquels, ne peuvent pas partir plusieurs transitions du même état avec le même caractère et n'accepte pas d'ε-transition.

Les automates à états finis (2)

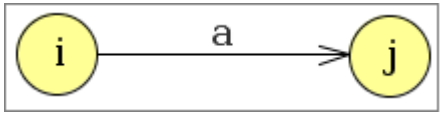
- **Automates finis non déterministes (AFN) :**
- Un automate fini non déterministe noté **AFN**, noté par l'expression aut = <Σ,E,D,F,T> avec :
  - Un ensemble Σ de symboles d'entrée, l'alphabet du langage. On considère que la chaîne vide ε, n'est jamais un membre de Σ.
  - L'ensemble fini E d'états.
  - Une fonction de transition qui donne pour chaque état et pour chaque symbole de ΣU{ε}, l'ensemble des états suivants T, sous ensemble de ExΣxE.
  - L'ensemble D sous-ensemble de E, des états de départ.
  - L'ensemble des états F, sous-ensemble de E, l'ensemble des états d'acceptation ou états finaux ou états terminaux.

Les automates à états finis (3)

- **Automates finis non déterministes (AFN) :**
  - Exemple :
    - Σ = {a,b,c} ; E={1, 2, 3, 4} ; D={1} ; F = {3,4} , T ={(1,a,2), (2,b,3),(1,c,4),(3,c,2)}

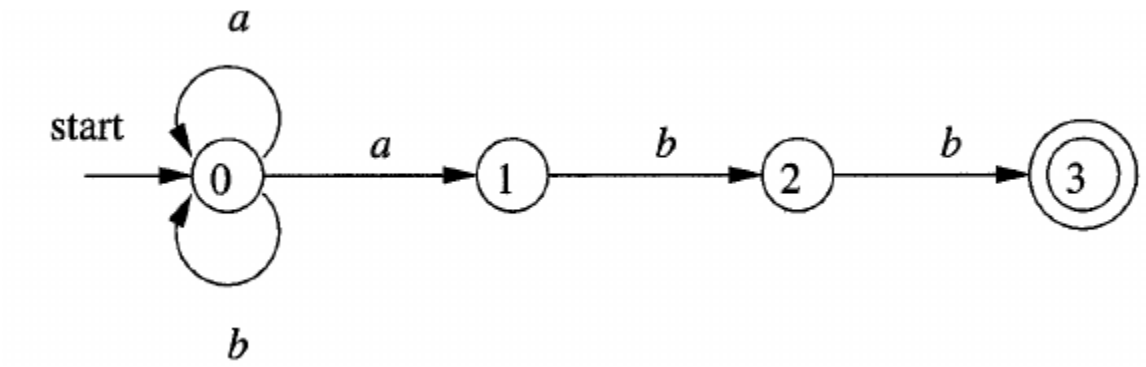
Les automates à états finis (4)

- **Automates finis non déterministes (AFN) :**
- **Graphe d'états :**
  - Un graphe d'états ressemble à un diagramme de transition, mais dans le cas des AFN, le même caractère peut étiqueter deux transitions ou plus en sortie d'un même état et les arcs peuvent être étiquetés par le même symbole spéciale ε.
  - Tous les états de départ sont représentés par un cercle où apparaît une flèche entrante :
  - Les états terminaux sont représentés par un cercle double
  - S'il existe un arc étiqueté par a entre l'état i et l'état j, si le triplet (i, a, j) ϵ T on dessine cet arc comme suite :



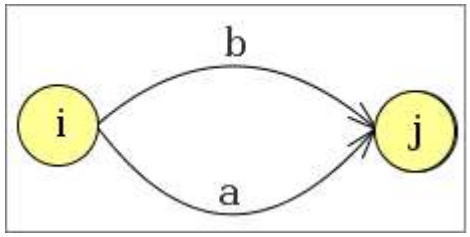
Les automates à états finis (5)

- **Automates finis non déterministes (AFN) :**
- **Graphe d'états :**
  - **Exemple :** soit le langage (a|b)*abb ; Σ={a,b} ; E={0,1,2,3} ; D={0} ; F={3} ; T={(0,a,0),(0,b,0),(0,a,1),(1,b,2),(2,b,3)}

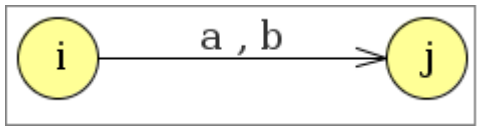


Les automates à états finis (6)

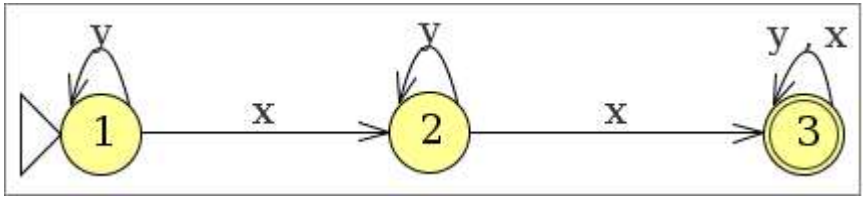
- **Automates finis non déterministes (AFN) :**
- **Graphe d'états :**
- **Notation :**



est équivalent à



- si tout état d'un ensemble donné Σ étiquette des transitions d'un état i vers un état j on notera Σ={a, b, c}
- **Exemple :** Σ = {x, y} ; E ={1, 2 ,3} ; D ={1} ; F ={3} ; T ={(1, y, 1), (1, x, 2), (2, y, 2), (2, x, 3), (3, x, 3), (3, y, 3)}



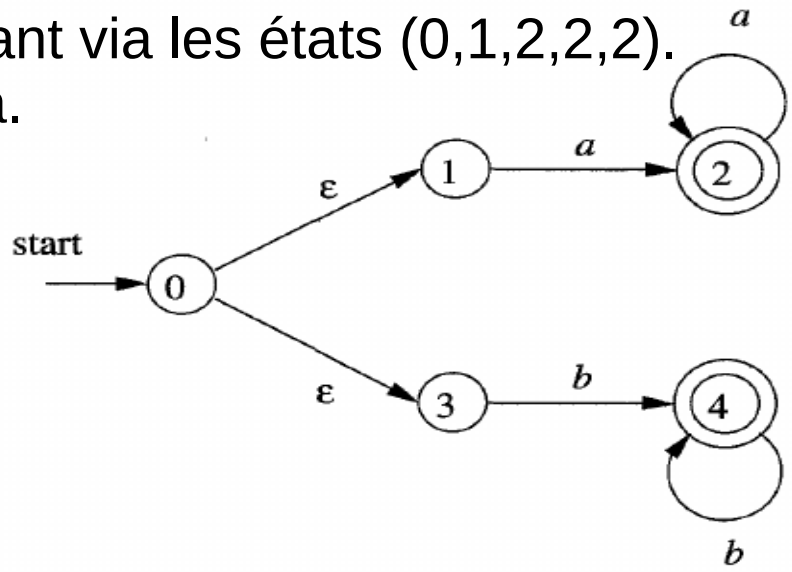
Les automates à états finis (7)

- **Table de transition :**
  - Une autre méthode de représentation d'un automate est la table des transitions, pour décrire la fonction de transition T on utilisera une table M indicée par E et Σ, tel que si e ϵ E, a ϵ Σ, e' ϵ M(e,a) ssi (e, a, e') ϵ T
  - **Exemple :** table de transition correspondante à l'expression régulière (a|b)*abb

État	a	b	ε
0	{0,1}	{0}	-
1	-	{2}	-
2	-	{3}	-
3	-	-	-

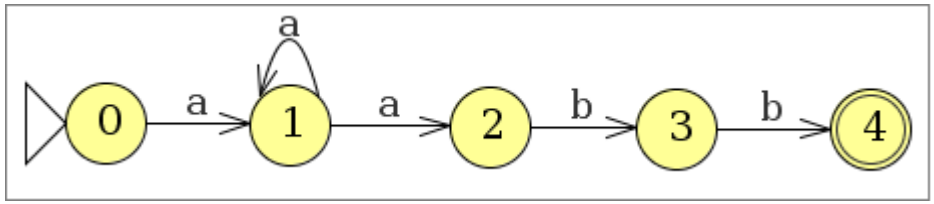
Les automates à états finis (8)

- **Acceptation d'une chaîne par un automate :**
  - Un AFN accepte une chaîne d'entrée x ssi il existe un chemin dans le graphe de transition entre l'état de départ et un état d'acceptation (final), tel que les étiquettes des arcs le long de ce chemin épellent (enlever le poile) la chaîne x.
  - **Exemples** : La chaîne aabb est acceptée par l'automate de l'exemple (slide 49) ,en se déplaçant via les états (0,0,1,2,3).
  - Soit l'automate suivant qui accepte les mots du langage engendrés par l'expression régulière aa*|bb*
  - La chaîne aaa est acceptée en se déplaçant via les états (0,1,2,2,2). les étiquettes de ces arcs sont εaaa = aaa.

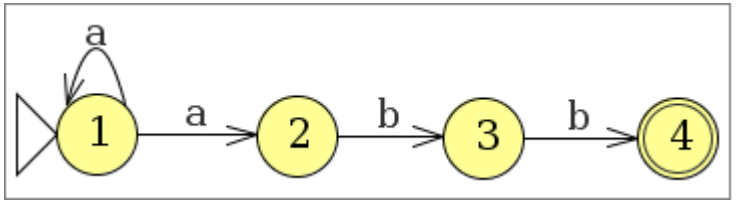


Les automates à états finis (9)

- **Acceptation d'une chaîne par un automate :**
- **Exemples (suite) :** soit l'expression régulière a+abb et la chaîne x=abb, l'AFN correspondant à cette expression est :



  - Cet automate n'accepte pas la chaîne x.
  - Si on considère a*abb, l'automate correspondant est :



  - Cet automate accepte la chaîne x=abb en parcourant le chamin 1,2,3,4.

Les automates à états finis (10)

- **Automates finis déterministes :**
- Un automate fini déterministe (AFD) est un cas particulier d'un automate fini non déterministe (AFN) dans lequel :
  - Aucun état n'a une ε-transition (une transition sur l'entrée ε).
  - Pour chaque état e et chaque symbole a, il existe un et un seul arc étiqueté a qui quitte e.
- Un AFD a au plus une transition à partir de chaque état sur n'importe quel symbole, en conséquence il est très facile de déterminer si un AFD accepte une chaîne d'entrée, étant donné qu'il existe au plus un chemin depuis l'état de départ étiqueté par cette chaîne.
- Si on utilise une table de transition pour représenter un AFD, chaque entrée est un état singleton.
- Un AFN est une représentation abstraite d'un algorithme de reconnaissance des chaînes d'un langage.
- Un AFD est un algorithme concret de reconnaissance de chaînes.
- N'importe quel AFN ou expression régulière peuvent être transformées en un AFD.

Les automates à états finis (11)

- **Simulation d'un AFD :**
- Soit x une chaîne d'entrée qui se termine par un caractère de fin de fichier (fdf ou eof), soit e_0 un état de départ et F l'ensemble des états terminaux.
- Soit la fonction transiter(e,c) qui donne l'état vers lequel il y a une transition depuis l'état e sur le caractère d'entrée c, la fonction carsuiv(x) retourne le prochain caractère de la chaîne d'entrée x. La simulation d'un AFD est donnée par l'algorithme suivant :

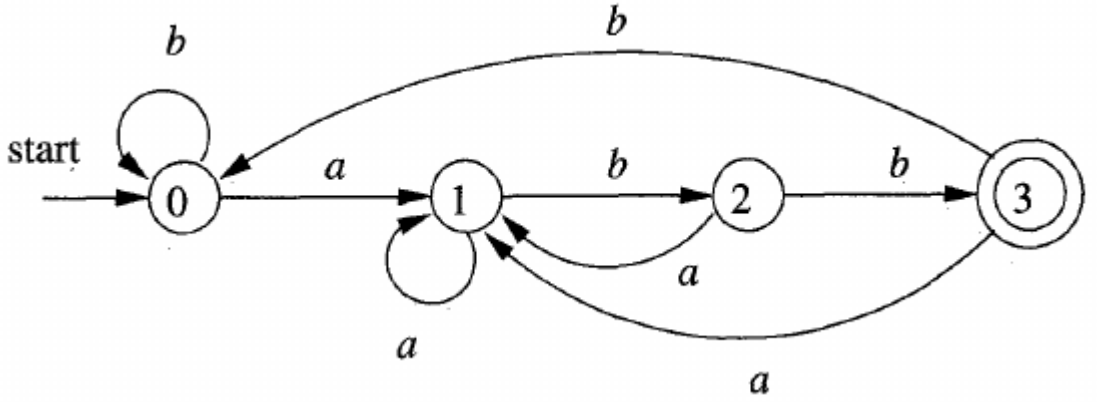
```
e=e0
c=carsuiv(x)
tant que (c!=fdf){
   e = transiter(e,c)
   c = carsuiv(x)
}
si (e  ϵ F) alors retrouner ''oui''
sinon retourner ''non''
```

Les automates à états finis (12)

- **Simulation d'un AFD :**
- **Exercice d'application :**
  - dessiner l'AFD correspondant au langage L((a|b)*abb), cette automate accepte-t-il la chaîne x=ababb.

Les automates à états finis (13)

- **Simulation d'un AFD :**
- **Exercice d'application :** dessiner l'AFD correspondant au langage L((a|b)*abb), cette automate accepte-t-il la chaîne x=ababb.
- **Solution :** l'automate correspondant au langage L((a|b)*abb)) est :



- Cet automate accepte la chaîne x en parcourant les états 0,1,2,1,2,3

Automates à états finis aux expressions régulières

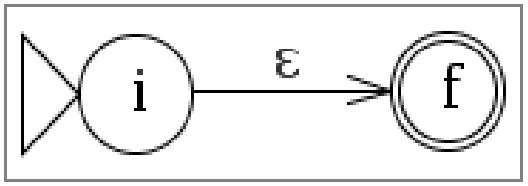
- **Transformation d'une expression régulière en un AFN :**
  - Il existe de nombreuses stratégies pour construire un reconnaisseur à partir d'une expression régulière.
  - Un stratégie qui a été utilisée dans un grand nombre d'analyseurs lexicaux, consiste à construire un AFN à partir d'une expression régulière et ensuite convertir l'AFN en AFD.

Automates à états finis aux expressions régulières

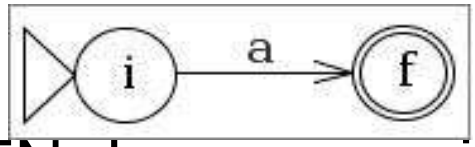
- **Transformation d'une expression régulière en un AFN :**
- **Algorithme de McNaughton-Yamada-Thompson :**
  - **Donnée** : une expression régulière *r* sur l'alphabet Σ.
  - **Résultat** : un AFN *N* qui accepte *L(r)*.
  - **Méthode** : on décompose r en sous expressions régulières, on construit les AFN pour chacun des symboles de base de r (soit ε soit les symboles de Σ). on se guidant de la structure de r on combine récursivement ces AFN en utilisant les règles de construction ci-dessous, jusqu'à l'obtention de l'AFN de l'expression complète.

Automates à états finis aux expressions régulières

- **Transformation d'une expression régulière en un AFN :**
- **Algorithme de McNaughton-Yamada-Thompson :**
  - **Règles :**
    - L'AFN correspondant à ε est :

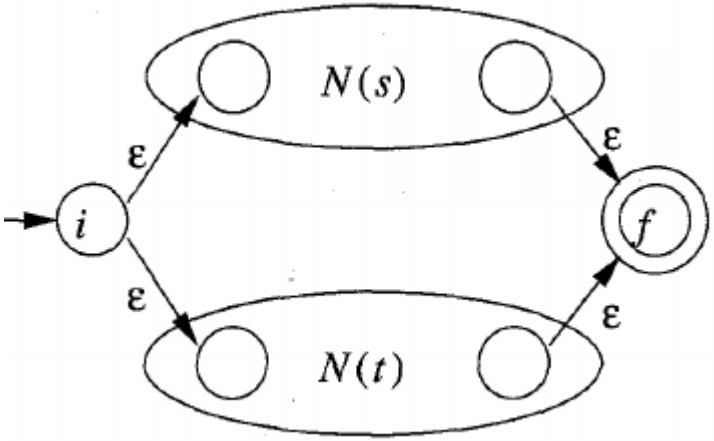


    - Pour chaque symbole a dans Σ construire l'AFN :



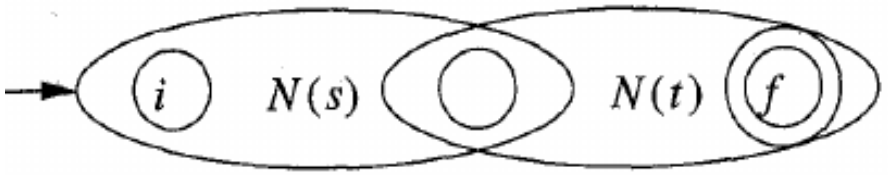
    - On suppose que les AFN des expressions régulières *s* et *t* sont N(s) et N(t), l'AFN correspondant à l'expression régulière r=s|t est le suivant :

Remarque : Les états de départ et d'acceptation de N(s) et N(t) ne sont pas les états de départ et d'acceptation de N(s|t). Ils sont donc des états ordinaires de ce dernier.



Automates à états finis aux expressions régulières

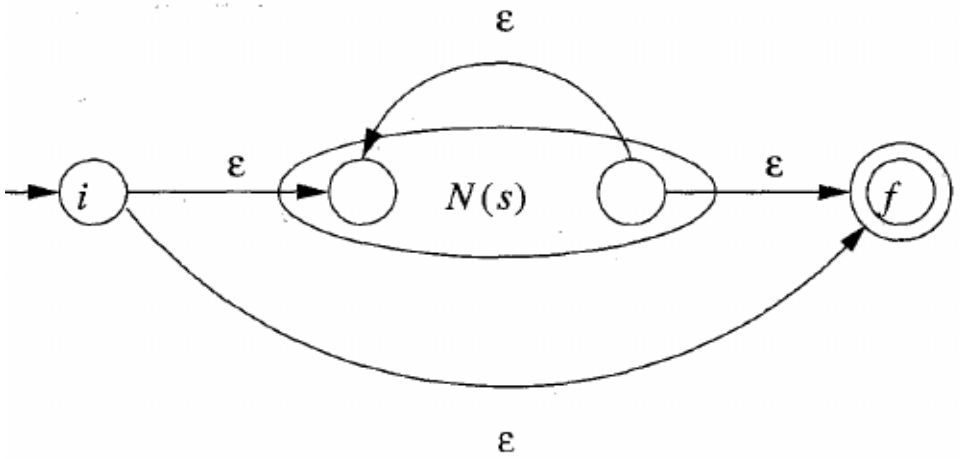
- **Transformation d'une expression régulière en un AFN :**
- **Algorithme de McNaughton-Yamada-Thompson :**
  - **Règles :**
    - L'AFN correspondant à r=st est le suivant :



    - l'état de départ de N(s) devient l'état de départ de l'AFN composé et l'état d'acceptation de N(t) devient l'état d'acceptation de N(r).

Automates à états finis aux expressions régulières

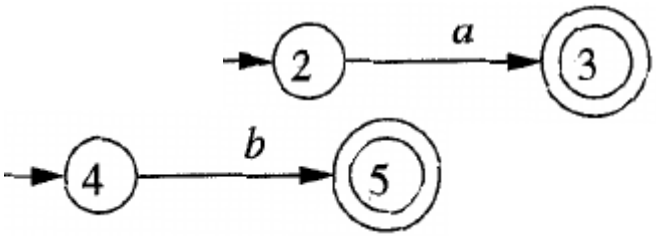
- **Transformation d'une expression régulière en un AFN :**
- **Algorithme de McNaughton-Yamada-Thompson :**
  - **Règles :**
    - L'AFN correspondant à l'expression régulière r=s* est donné par :



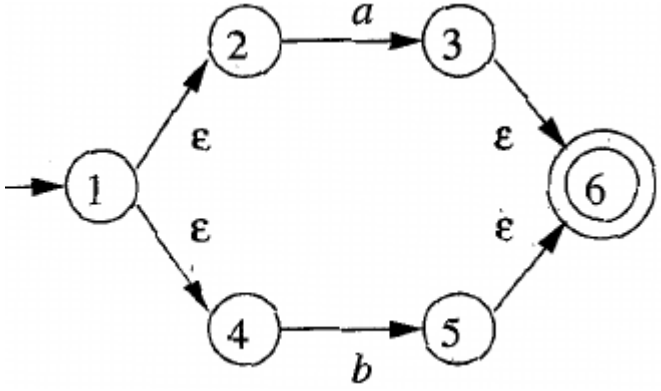
    - On peut aller de i à f directement en suivant un arc étiqueté ε qui représente le fait que ε ϵ (L(s))* ou bien en traversant N(s) une ou plusieurs fois. Pour l'expression régulière (s) l'AFN correspondant est le même pour l'expression régulière s.

Automates à états finis aux expressions régulières

- **Transformation d'une expression régulière en un AFN :**
- **Algorithme de McNaughton-Yamada-Thompson :**
  - **Exemple :** (a|b)*abb
    - r1=a
    - r2=b

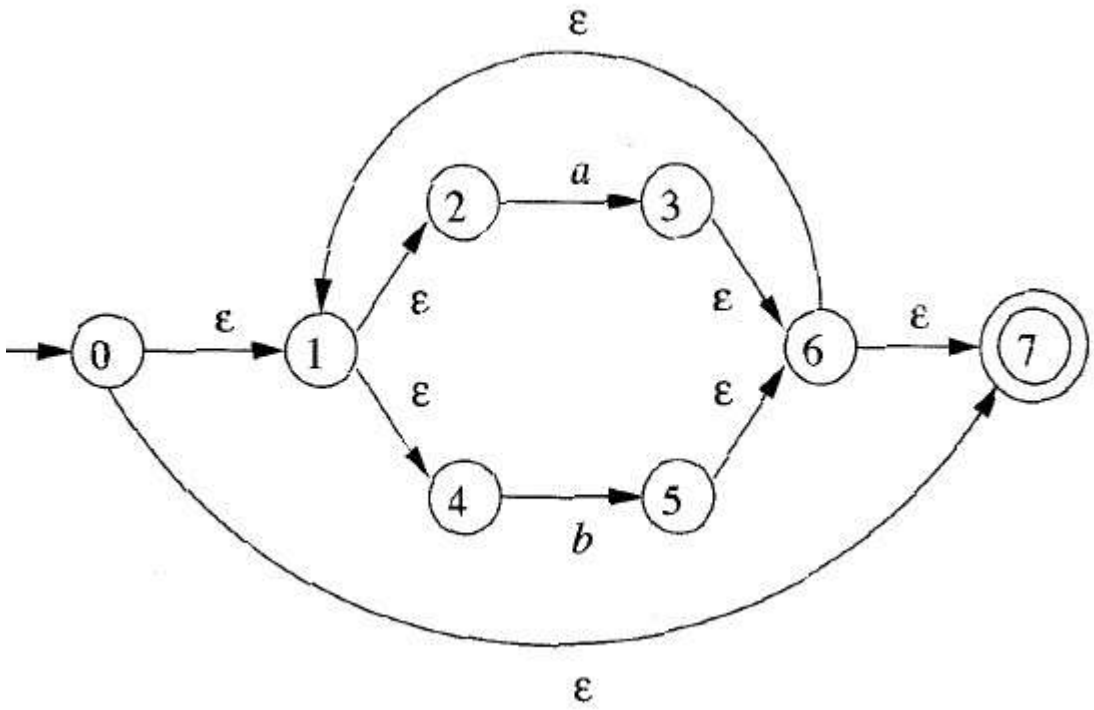


    - Maintenant on peut combiner N(r1) et N(r2), pour obtenir l'AFN N(r3) pour r3=r1|r2



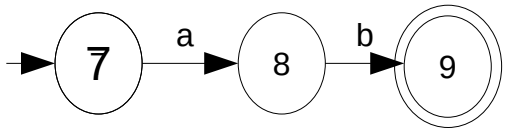
Automates à états finis aux expressions régulières

- **Transformation d'une expression régulière en un AFN :**
- **Algorithme de McNaughton-Yamada-Thompson :**
  - **Exemple :** (a|b)*abb
    - r4=(r3)* donc l'AFN N(r4) est

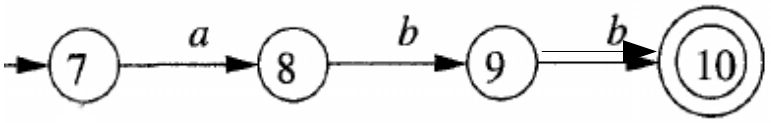


Automates à états finis aux expressions régulières

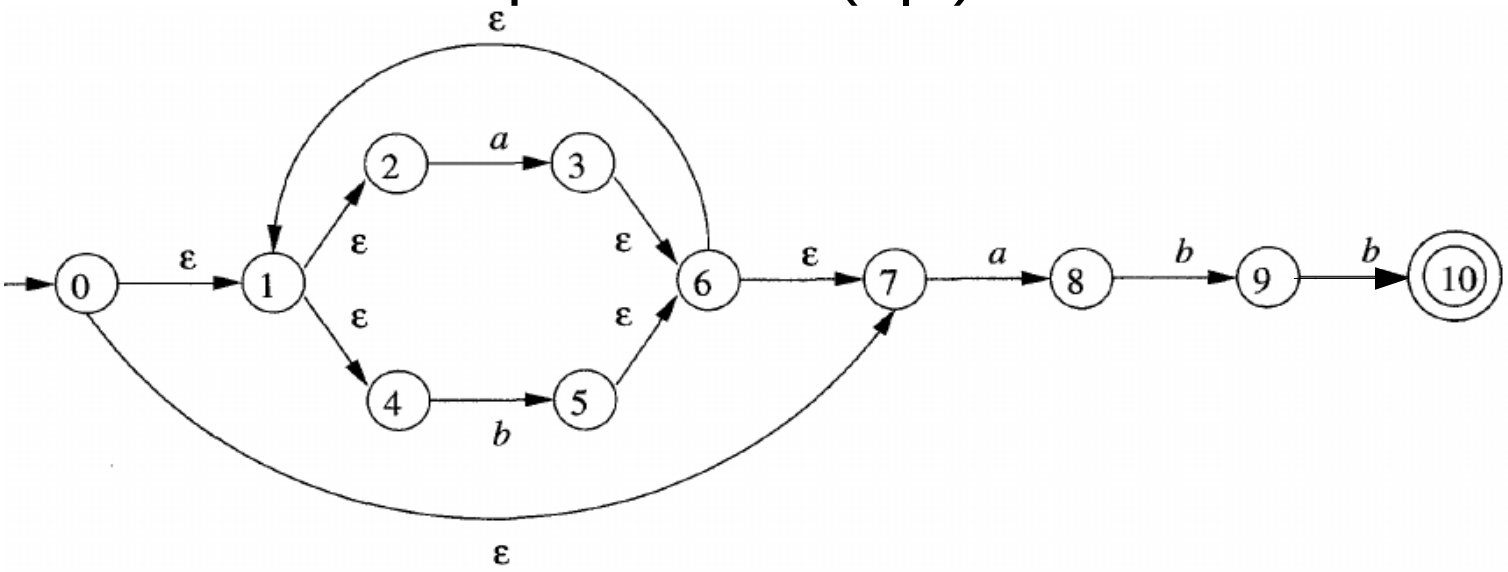
- **Transformation d'une expression régulière en un AFN :**
- **Algorithme de McNaughton-Yamada-Thompson :**
  - **Exemple :** (a|b)*abb
    - r5=r1r2 l'AFN correspondant à r5 est :



    - r6=r5r2 l'AFN correspondant à r6 est :

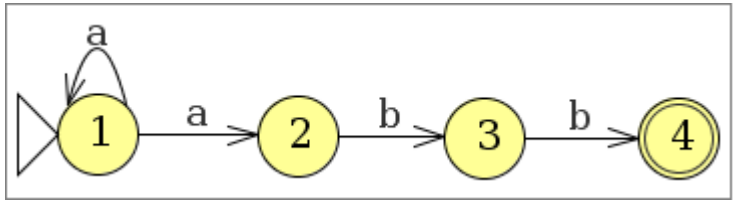


    - r=r4r6 l'AFN correspondant à (a|b)*abb est :

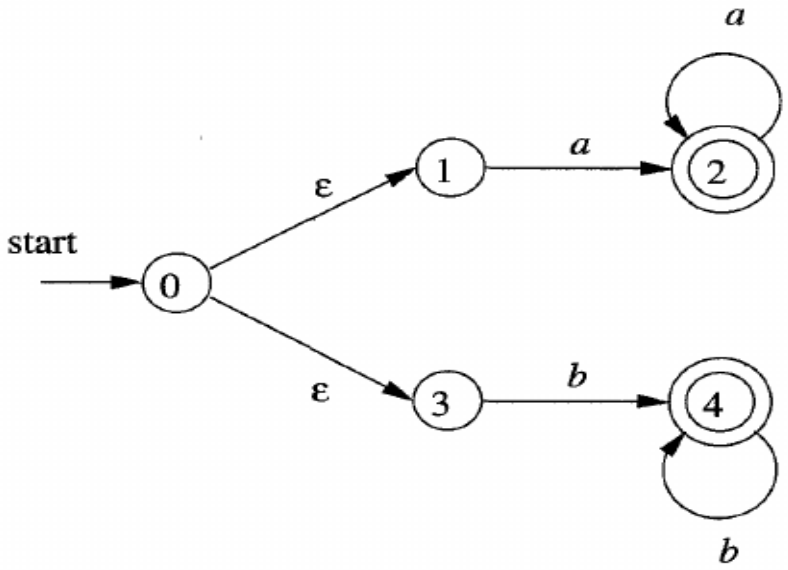


Automates à états finis aux expressions régulières

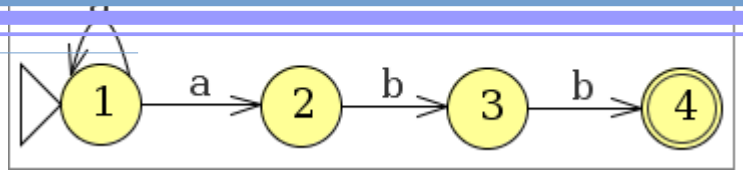
- **Transformation d'un AFN en un AFD :**
- Soit les automates suivants :
- Aut1



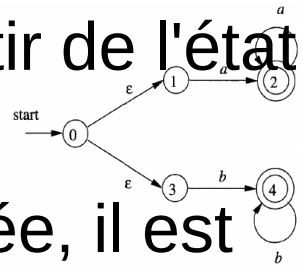
- Aut2 :



Automates à états finis aux expressions régulières



- **Transformation d'un AFN en un AFD :**
- L'automate 1 a deux transitions de l'état 1 sur la chaîne d'entrée a qui signifie qu'on peut aller vers l'état 1 ou 2.
- De même l'automate 2 a deux transitions sur ε à partir de l'état 0.



- Dans le cas où la fonction de transition est multivaluée, il est difficile de simuler un AFN à l'aide d'un programme.
- L'acceptation d'une chaîne d'entrée suppose qu'il puisse exister un chemin étiqueté par cette chaîne qui mène depuis l'état de départ jusqu'à un état d'acceptation.
- Mais il existe dans un AFN plusieurs chemins qui épellent la même chaîne d'entrée, on doit les prendre en considération avant de décider d'accepter ou non la chaîne d'entrée.

Automates à états finis aux expressions régulières

- **Transformation d'un AFN en un AFD :**
- L'idée générale de la transformation d'un AFN en un AFD est qu'un état de l'AFD correspond à un ensemble d'états de l'AFN.
- L'AFD utilise un état pour garder trace de tout les états possibles que l'AFN peut atteindre après avoir lu chaque symbole d'entrée.

Automates à états finis aux expressions régulières

- **Transformation d'un AFN en un AFD :**
- Algorithme :
  - Donnée : un AFN
  - Résultat : un AFD qui accepte le même langage de l'AFN.
  - Méthode : l'algorithme construit une table de transition Dtrans pour l'AFD.
  - Chaque état de l'AFD est un ensemble d'états de l'AFN. On construit Dtrans de telle manière que l'AFD simulera tous les déplacements possibles que l'AFN peut effectuer sur une chaîne d'entrée donnée.

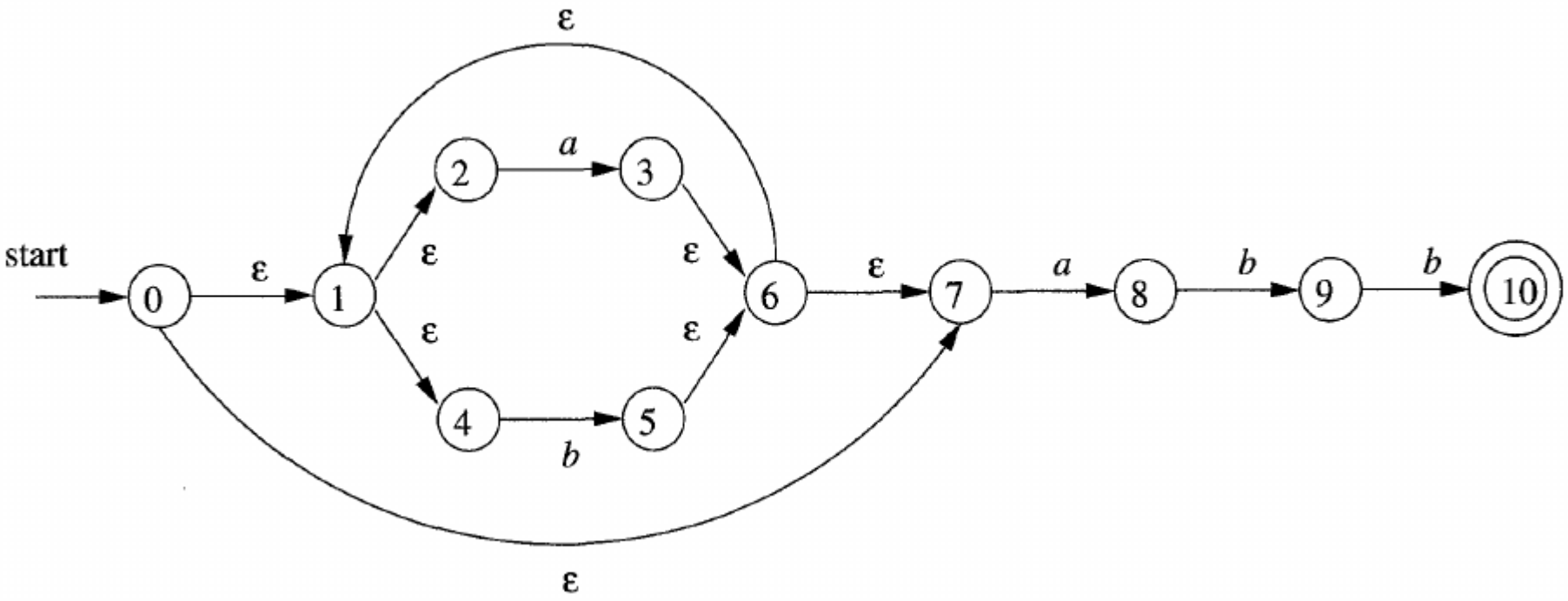
Automates à états finis aux expressions régulières

- **Transformation d'un AFN en un AFD :**
- Algorithme :
  - Pour garder trace des ensemble des états de l'AFN on utilisera les opérations suivantes :
  - Soit e et T tels que e représente un état de l'AFN et T représente un ensemble d'états de l'AFN.

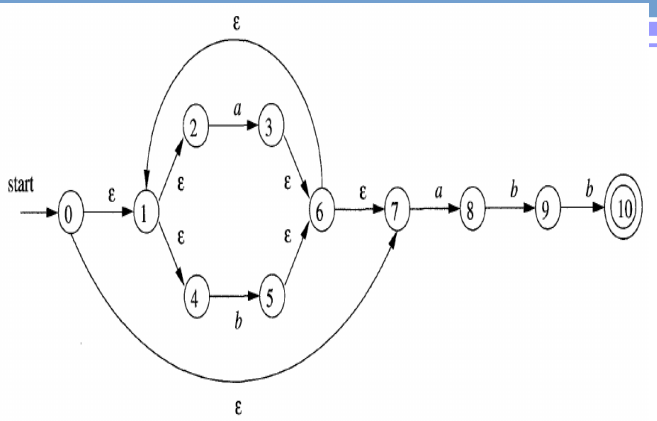
Opération	Description
ε-fermeture(e) Noté aussi ε-f(e)	Ensemble des états de l'AFN accessibles depuis l'état e de l'AFN par des ε-transition uniquement.
ε-fermeture(T) Noté aussi ε-f(T)	Ensemble des états de l'AFN accessibles depuis un état e ϵ T par des ε-transition uniquement.
transiter(T,a) Noté aussi tran(T,a)	Ensemble des états de l'AFN vers lesquels il existe une transition sur a

Automates à états finis aux expressions régulières

- **Transformation d'un AFN en un AFD :**
- Exemple d'application : soit l'AFN suivant qui accepte le langage (a|b)*abb

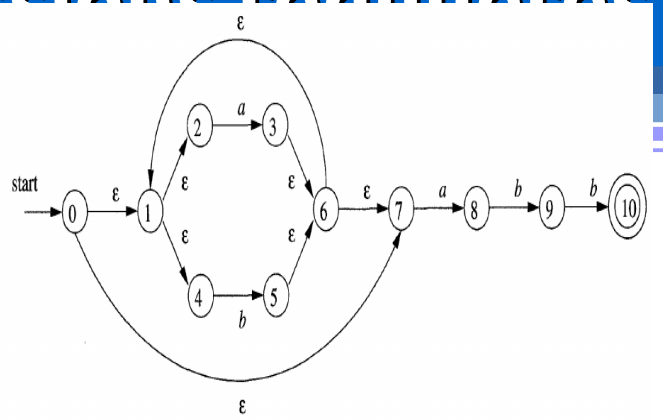


Automates à états finis aux expressions régulières

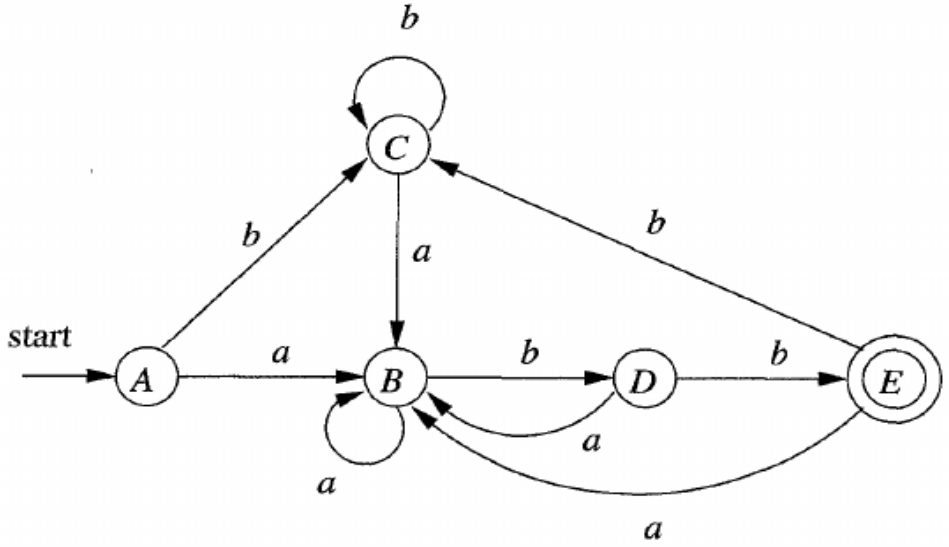


- **Transformation d'un AFN en un AFD :**
- Exemple d'application :
- L'état de départ de l'AFD A=ε-f(0)={0,1,2,4,7}
- L'alphabet Σ = {a, b}
- B=ε-f(tran(A,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(A,a)=B.
- C=ε-f(tran(A,b))=ε-f({5})={1,2,4,5,6,7} => Dtrans(A,b)=C.
- ε-f(tran(B,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(B,a)=B.
- ε-f(tran(B,b))=ε-f({5,9})={1,2,4,5,6,7,9} => Dtrans(B,b)=D.
- ε-f(tran(C,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(C,a)=B.
- ε-f(tran(C,b))=ε-f({5})={1,2,4,5,6,7} => Dtrans(C,b)=C.
- ε-f(tran(D,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(D,a)=B.
- ε-f(tran(D,b))=ε-f({5,10})={1,2,4,5,6,7,10} => Dtrans(D,b)=E.
- ε-f(tran(E,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(E,a)=B.
- ε-f(tran(E,b))=ε-f({5})={1,2,4,5,6,7} => Dtrans(E,b)=C.

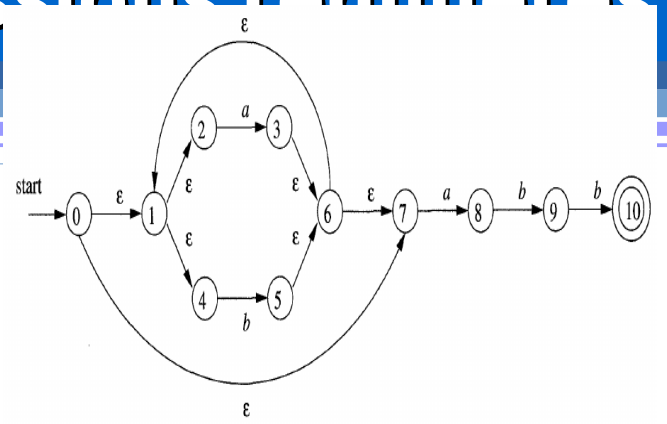
Automates à états finis aux expressions régulières



- **Transformation d'un AFN en un AFD :**
- Exemple d'application :
- L'état E est l'état de fin de l'AFD car E contient l'état 10 qui est l'état d'acceptation de l'AFN.
- **Règle :** un état de l'AFD est un état d'acceptation si c'est un ensemble d'états de l'AFN qui contient au moins un état d'acceptation de l'AFN.
- L'AFD équivalent à l'AFN est

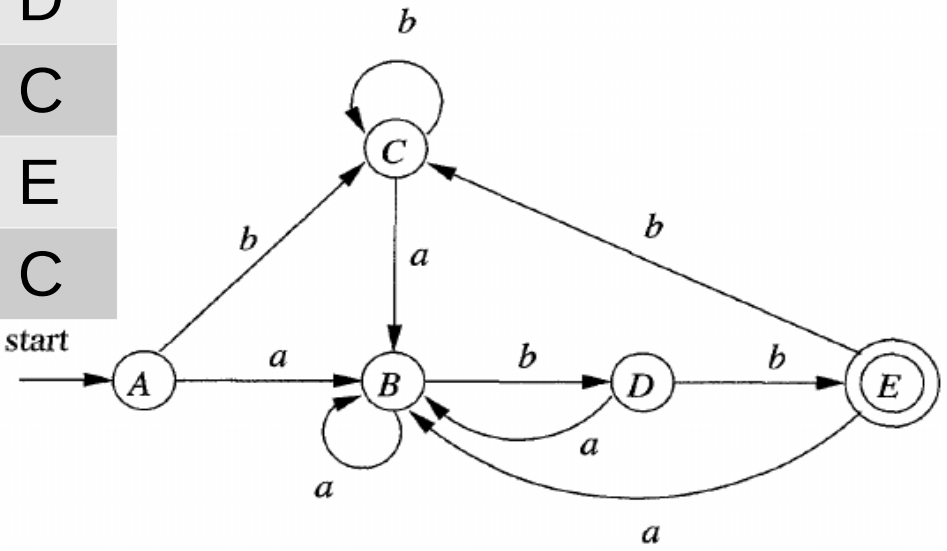


Automates à états finis aux expressions régulières



- **Transformation d'un AFN en un AFD :**
- Exemple d'application :
- Table de transition de l'AFD :

États de l'AFN	États de l'AFD	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,3,5,6,7,10}	E	B	C



Automates à états finis aux expressions régulières

- **Simulation d'un AFN :**
  - **Donnée** : Étant donné une chaîne x se terminant par un caractère de fdf (eof), un AFN N disposant d'un état de départ 0 et d'un ensemble d'états d'acceptation F et une fonction transiter.
  - **Résultat** : réponse oui si N accepte la chaîne et non autrement.
  - **Méthode** : l'algorithme suivant appliqué à x permet de renvoyer oui si l'AFN accepte x et non autrement.

Automates à états finis aux expressions régulières

- **Simulation d'un AFN :**

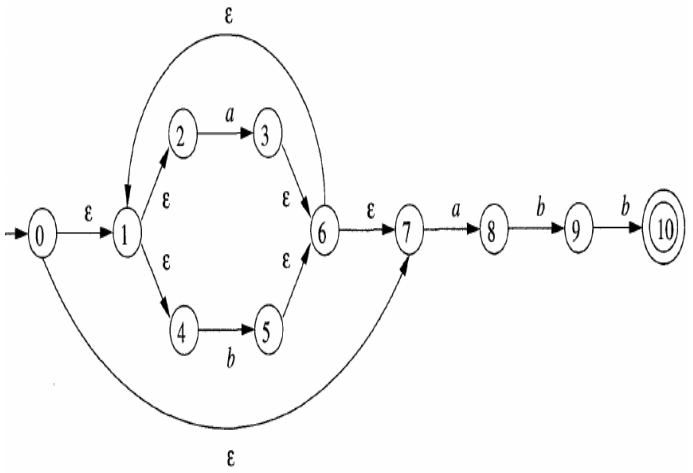
```
E =  ε-f({0})
c = carSuiv(x)
tant que (a!=fdf){
   E =  ε-f(transiter(E,c))
   c = carSuiv(x)
}
si(E ∩ F!=Ø) alors retourner oui
sinon retourner non
```

  - L'algorithme réalise la construction des sous-ensembles à l'exécution, il calcule une transition depuis l'ensemble courant d'états E vers le prochain ensemble d'états. Implicitement l'algorithme transforme l'AFN en AFD et détermine si l'AFD accepte-t-il la chaîne d'entrée x.

Automates à états finis aux expressions régulières

- **Simulation d'un AFN :**
- **Exemple :** Soit l'AFN correspondant à (a|b)*abb et la chaîne d'entrée x = aabb

E = ε-f({0})={0,1,2,4,7}
c = carSuiv(x)=a
E = ε-f(transiter(E,a))={3,1,2,4,6,7,8}
carSuiv(x)=a
E = ε-f(transiter(E,a))={3,1,2,4,6,7,8}
carSuiv(x)=b
E = ε-f(transiter(E,b))={1,2,4,5,6,7,9}
carSuiv(x)=b
E = ε-f(transiter(E,b))={1,2,4,5,6,7,10}
carSuiv(x)=fdf
E ∩ F =10 alors l'AFN accepte la chaîne x=aabb



Minimisation du nombre des états d'un AFD

- Étant donné que deux AFD peuvent reconnaître les mots du même langage, le but de la minimisation du nombre d'états d'un AFD consiste à produire un AFD avec le minimum d'états possibles.
- L'algorithme de minimisation consiste à partitionner les états de l'AFD en des groupes d'états qui ne peuvent pas être distingués. Chaque groupe d'états qui ne peuvent pas être distingués est alors fusionné en un état unique.
- Initialement la partition consiste en deux groupes :
  - Les états d'acceptation et les états de non-acceptation.
  - L'étape fondamentale consiste à prendre un groupe d'état $A=\{e_1, e_2, \ldots, e_k\}$ et un symbole d'entrée 'a' et regarder si 'a' peut être utilisé pour distinguer des états du groupe A.
  - On examine les transitions depuis chaque état $e_1, e_2, \ldots, e_k$ sur l'entrée 'a', si ces transitions conduisent à des états qui tombent au moins dans deux groupes différents de la partition courante, alors, on doit diviser A en une collection de groupes, de sorte que e_i et e_j sont dans le même groupe ssi ils partent vers le même groupe sur le symbole 'a'.
  - On répète le processus de division de groupe de la partition courante jusqu'à ce que aucun groupe n'est besoin d'être divisé.

Minimisation du nombre des états d'un AFD

- **Algorithme de minimisation des états d'un AFD :**
  - **Donnée :** un AFD D avec un ensemble d'états E, un alphabet d'entrée Σ, un état de départ e_0 et en ensemble d'états d'acceptation F.
  - **Résultats :** un AFD D' qui reconnaît le même langage que D et ayant le peu d'états possibles.

Minimisation du nombre des états d'un AFD

- **Algorithme de minimisation des états d'un AFD :**
  - **Méthode :**

    1) Construire une partition initiale Π de l'ensemble des états avec deux groupes ; les états d'acceptation F et les états de non-acceptation E-F.

    2) Appliquer la procédure suivante à Π pour construire une nouvelle partition Π_{new}

    soit $\Pi_{new} = \Pi$

    pour chaque groupe G de Π {

    - Partitionner G en sous groupes de manière que deux états s et t de G soient dans le même sous groupe ssi pour tout symbole d'entrée 'a', les états s et t ont des transitions sur 'a' vers les états du même groupe de Π.
    - Remplacer G dans Π_{new} par les sous groupes formés.}

    3) si $\Pi_{new} = \Pi$ alors $\Pi_f = \Pi$ aller à 4 sinon répéter l'étape 2 avec $\Pi = \Pi_{new}$

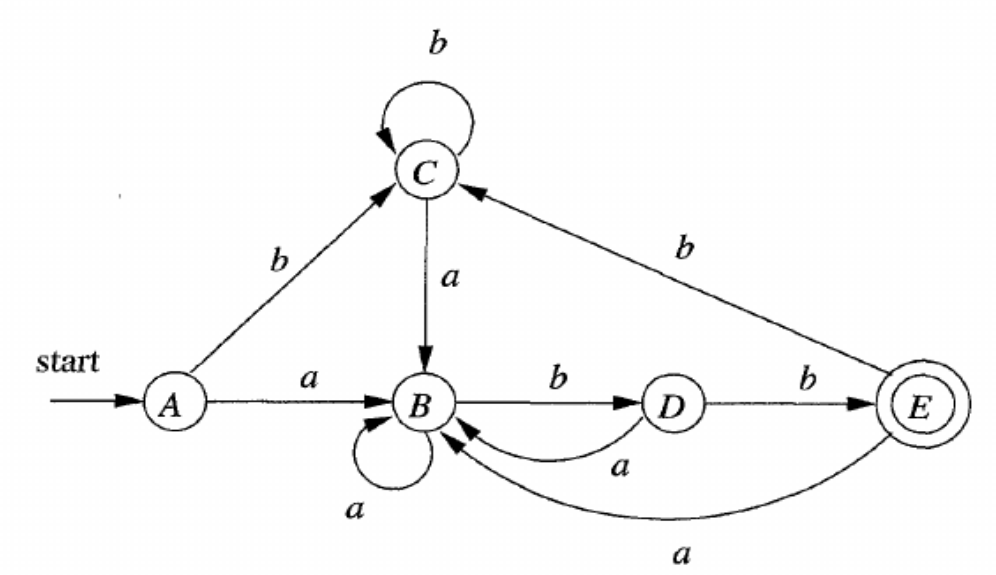
    4) Choisir un état de chaque groupe de la partition Π_f en tant que représentant de ce groupe. Les représentants seront les états de l'AFD réduit D'.

    - L'état de départ de D' est le représentant du groupe qui contient l'état de départ e_0 de D.
    - Les états de d'acceptation de D' sont les représentants contenant un état d'acceptation de D.

    5) Si D' contient un état mort, c-à-d un état e qui n'est pas un état d'acceptation et qui a des transitions vers lui-même sur tout les symboles d'entrée, alors supprimer e de D'.

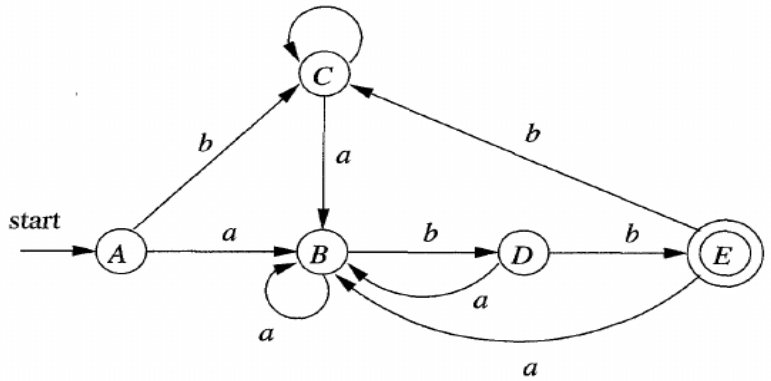
Minimisation du nombre des états d'un AFD

- **Exemple :** soit l'AFD correspondant au langage L((a|b)*abb)



Minimisation du nombre des états d'un AFD

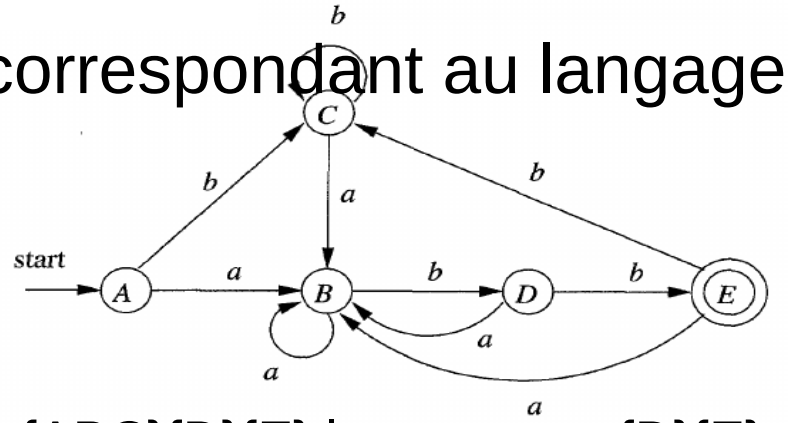
- **Exemple :** soit l'AFD correspondant au langage L((a|b)*abb)



- La partition initiale Π est constituée de deux groupes, {E}{ABCD} le groupe des états d'acceptation, et le groupe états de non-acceptation.
- Pour construire la partition Π_{new} le groupe {E} est constitué d'un seul état donc, il ne peut pas être découpé.
- Le groupe {ABCD} sur le symbole *a* chacun des 4 états a une transition vers B, ce qui ne les distingue pas.
- Sur le symbole *b* A,B et C ont une transition vers les états du groupe {ABCD}, tandis que D a une transition vers E qui est membre d'un autre groupe.
- Dans la partition Π_{new} le groupe {ABCD} doit être découpé en 2 nouveaux sous groupes {ABC} {D}. donc dans l'étape suivante la partition Π_{new} = {ABC}{D}{E}.

Minimisation du nombre des états d'un AFD

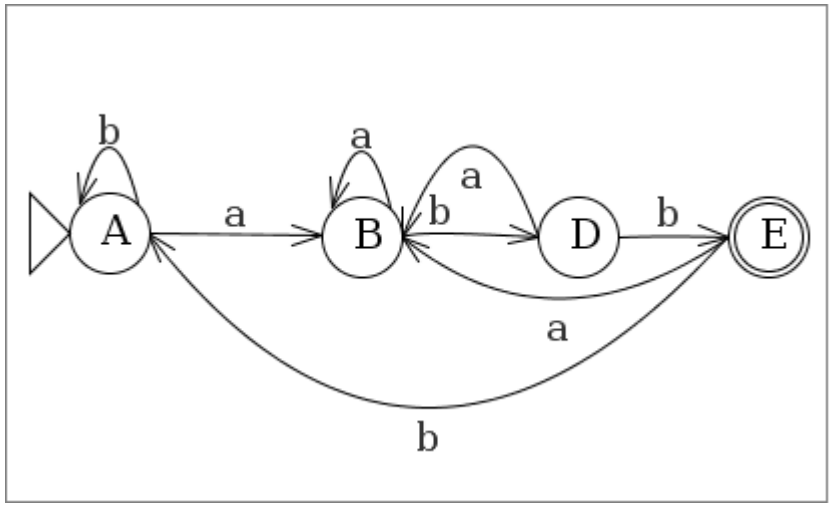
- **Exemple :** soit l'AFD correspondant au langage L((a|b)*abb)



- Retour à 2 avec la partition {ABC}{D}{E} les groupes {D}{E} sont singleton, donc ils ne peuvent pas être découpés.
- Sur le symbole *a* aucun découpage. Sur le symbole *b*, A et C ont une transition sur B tandis que B a une transition vers D membre d'un autre groupe. D'où Π_{new} = {AC}{B}{D}{E}.
- Retour à 2 avec la nouvelle partition :
  - Sur le symbole *a*, A et C conduisent vers le même état B.
  - Sur le symbole *b*, A et C conduisent vers le même état C.
  - Par conséquent A et C constituent le même groupe d'où la partition finale Π_f ={AC}{B}{D}{E}.

Minimisation du nombre des états d'un AFD

- **Exemple :** soit l'AFD correspondant au langage L((a|b)*abb)
- Si on choisit A comme représentant du groupe {AC}, B, D et E pour les groupes singletons, on obtient alors l'automate réduit dont la table des transitions est :
- A est l'état de départ et E est l'état d'acceptation, le graphe d'état correspondant est :



État	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Le générateur d'analyseur lexicale Lex/Flex

- Lex/Flex est un outil de génération automatisée d'analyseur lexical, à partir de la spécification des expressions régulières pour décrire les modèles des unités lexicales.
- Lex/Flex transforme les modèles (expressions régulières et définitions régulières) en entrée en un diagramme de transition et génére le code source correspondant, dans un fichier appelé lex.yy.c, qui simule le diagramme de transition généré.
- Ce fichier écrit en langage C doit être compilé par le compilateur du C pour produire un exécutable qui sera l'analyseur lexical du langage décrit par les modèles en question.

Le générateur d'analyseur lexicale Lex/Flex

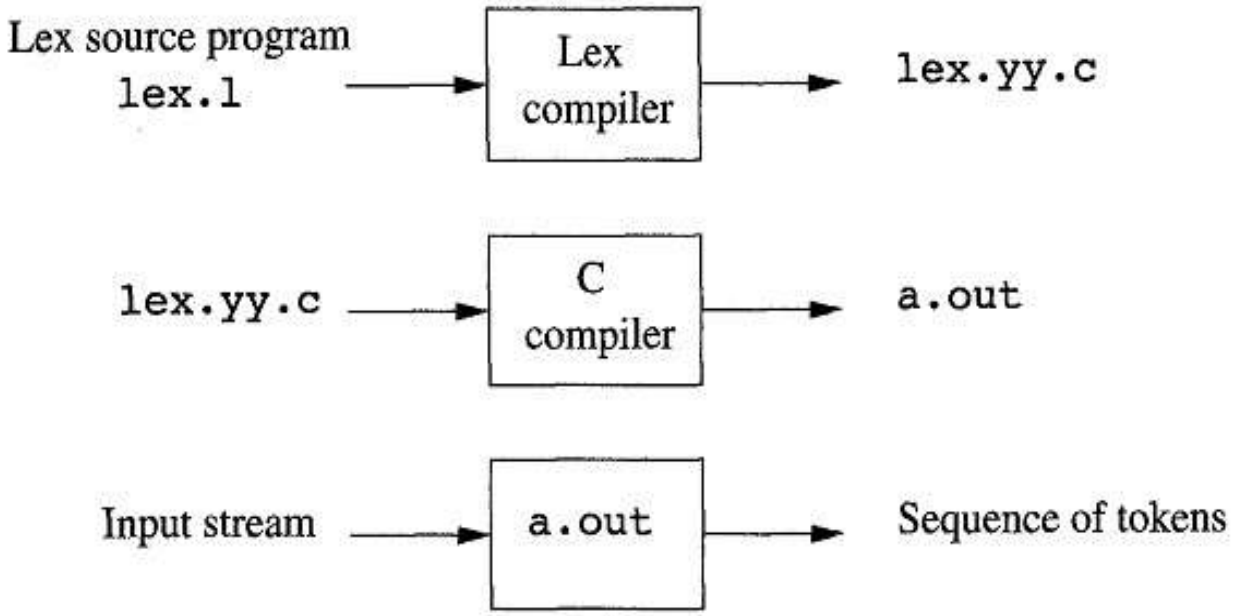
- Plusieurs langages dérivés de Lex existent :
  - Flex produit du code écrit en C/C++
  - Jlex/JFlex produit du code écrit en Java
  - Lecl produit du code écrit en Caml
  - ml-lex produit du code en ML
  - alex produit du code écrit en Haskell
  - tcl-tcLex produit du code écrit en tcl

Le générateur d'analyseur lexicale Lex/Flex

- **Utilisation de Lex :**
  - Un fichier ***lex.l*** écrit en langage Lex, qui décrit l'analyseur lexical à générer est présenté au compilateur Lex en entrée.
  - Le compilateur Lex transforme ***lex.l*** en un programme écrit en C dans un fichier appelé lex.yy.c.
  - Ce fichier est compilé par le compilateur C et produit un fichier exécutable ***a.out*** qui est le fichier du compilateur exécutable, il prendra en paramètre un fichier écrit dans le langage source et produira en sortie une séquence des unités lexicales.

Le générateur d'analyseur lexicale Lex/Flex

- **Utilisation de Lex :**



Le générateur d'analyseur lexicale Lex/Flex

- **Structure d'un programme Lex :**
  - Un fichier de description pour Lex est formé de trois parties, séparées par des lignes contenant seulement %%, aligné à gauche, selon le schéma suivant :

```
// Parties déclarations
%{ //(variables, constantes, bibliothèques C, etc.)
%}
//expressions régulières, définitions régulières
%%
// Partie productions (expressions régulières)
%%
// code de service (fonctions utilitaires)
```

Le générateur d'analyseur lexicale Lex/Flex

- **Structure d'un programme Lex :**
  - La partie déclarations comporte les déclarations des variables, des structures, des unions, des déclarations régulières, des déclaration des bibliothèques en C…
  - La partie productions, les déclarations prennent la forme suivante :

m_1	{action$_1$}
m_2	{action$_2$}
...	...
m_i	{action$_i$}
...	...
m_n	{action$_n$}

Les m_i sont des expressions régulières ou des définitions régulières de la partie déclaration.

Les action$_i$ sont les actions à réaliser par l'analyseur lexical si un lexème est accepté par m_i. Généralement écrites en langage C, mais d'autres langages peuvent être utilisés.

Le générateur d'analyseur lexicale Lex/Flex

- **Structure d'un programme Lex :**
  - La 3ème partie contient des fonctions qui peuvent être utilisées dans les actions de la 2ème partie.
  - Eventuellement ces fonctions peuvent être compilées séparément et chargées avec l'analyseur lexical.
  - Cette partie peut aussi contenir la fonction principale main pour une éventuelle exécution du programme produit.
- **Remarque** :
  - Un fichier source en Lex doit avoir l'extension .l (exemple.l).
  - Seulement la partie production est obligatoire dans Lex.

Le générateur d'analyseur lexicale Lex/Flex

- **Structure d'un programme Lex :**
  - **Exemple 1** : Un programme Lex qui affiche le contenu de son entrée standard vers l'écran.

```
%%
.|\n ECHO;
%%
```

  - **Exemple 2** : un programme Lex qui supprime tout les espaces et tabulations.

```
%%
[ \t] {/* supprime les espaces et tabulations */}
bye {exit(0);}
```

Le générateur d'analyseur lexicale Lex/Flex

- **Structure d'un programme Lex :**
  - **Exemple 3** : Un programme Lex qui reconnaît les verbes en anglais.
    - On considère la liste des verbes suivants :

is	am	are	were
was	be	being	been
do	does	did	will
would	should	can	could
has	have	had	go

Le générateur d'analyseur lexicale Lex/Flex

- **Structure d'un programme Lex :**
  - **Exemple 3** : Un programme Lex qui reconnaît les verbes en anglais.

```
%{
/*
* programme Lex qui reconnaît les verbes en Anglais
*/
%}
%%
[\t ]+      /* ignorer les espaces et tabulations */
is I
am I
are I
were I
was I
be I
beingbeen I
do I
does I
did I
will I
would I
should  I
can  I
could  I
has  I
have  I
had  I
go      {printf("%s :  is  a  verb\n", yytext);}
[a-zA-Z]+     {printf("%s:is  not  a verb\n", yytext);}
.|\n        {ECHO ; /* autres choses ou retour à la ligne */}
%%
main()
{
      yylex() ;
}
```

Le générateur d'analyseur lexicale Lex/Flex

- **Structure d'un programme Lex :**
  - **Remarques** :
    - yylex() : la fonction principale du programme écrit en LEX.
    - yytext : pointeur sur le début de la chaîne analysée (unité lexicale)
  - Exécution :
    - Lex : *lex verbes.l* produit un fichier lex.yy.c
    - gcc : gcc -o verbes lex.y.c -ll (ll : library lex)
    - Flex : *flex verbes.l* produit un fichier lex.yy.c
    - gcc : gcc -o verbes lex.yy.c -lfl (lfl : library fast lex)

Le générateur d'analyseur lexicale Lex/Flex

- **Structure d'un programme Lex :**
  - **Exercice d'application :** écrire un programme en lex qui reconnaît les identificateurs, mot clés et opérateurs en PASCAL

digit	→	`[0-9]`
digits	→	*digit*$^{+}$
number	→	*digits* (`.` *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

Le générateur d'analyseur lexicale Lex/Flex

```
%{
/*definitions des constantes
LT, LE, EQ, NE, GT, GE,
IF, THEN, ELSE, ID, NUMBER, RELOP*/
%}
/*definitions régulières*/
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number     (\+|\-)?{digit}+(\.{digit}+)?
(E[+-]?{digit}+)?
%%
{ws} {/*no action and no return*/}
if {printf("mot clé IF\n");}
then {printf("mot clé TEHN\n");}
else {printf("mot clé ELSE\n");}
{id}     {printf("identificateur     %s\
n",yytext);}
{number}       {printf("nombre       %s\
n",yytext);}
"<" {printf("relop LT\n");}
"<="   {printf("relop   LE\
n");}
"="    {printf("relop   EQ\
n");;}
"<>"   {printf("relop   NE\
n");}
">" {printf("relop GT\n");}
">="   {printf("relop   GE\
n");}
. {printf("non reconnue %s\
n",yytext);}
%%
int    main(int    argc,char
*argv[])
{
    yyin=fopen(argv[1],"r");
    yylex();
    fclose(yyin);
    return 0;
}
```