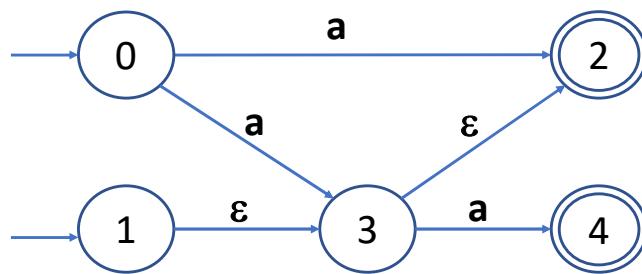


Automates finis non déterministes

- Un **AFND** est un **automate** qui vérifie au moins l'une de ces trois conditions :
 - Avoir plusieurs **états initiaux**.
 - Avoir des **transitions instantanées** (ou ε -**transitions**) : on peut passer d'un état à un autre sans rien lire (ce qui revient à lire ε).
 - Avoir au moins deux **transitions** qui sortent d'un même état « e » et avec une même étiquette « a ».
- Les **AFND** généralisent les **AFD**.

Exemple d'un AFND



Il y a 3 chemins de reconnaissance pour le mot a :

$(0, 2)$

$(0, 3, 2)$

$(1, 3, 4)$

Automates finis non déterministes

- Les **AFND** sont des **mauvais reconnaiseurs** : l'algorithme de reconnaissance par **AFND** est de **complexité exponentiel**.
- Pour reconnaître un mot w par un **AFND** :
 - On choisit un état initial « e_i ».
 - Ensuite, on applique itérativement :
 - on lit le prochain symbole « a » de w et on applique une **transition** avec l'étiquette « a » de l'état courant vers un état d'arrivée (il peut y avoir 0, 1 ou plusieurs).
 - on ne lit aucun symbole et on applique une ε -**transition**.
 - Jusqu'à atteindre un état final et la consommation de w .

Automates finis non déterministes

- Un **AFND** est un modèle mathématique M formé de cinq éléments :
 1. Σ : l'**alphabet** des symboles d'entrée
 2. E : un ensemble fini d'**états**
 3. δ : une **fonction de transition** définie de $E \times (\Sigma \cup \{\varepsilon\})$ vers $\mathcal{P}(E)$
 4. $I_0 \subseteq E$: l'ensemble des **états initiaux**
 5. $F \subseteq E$: l'ensemble des **états finaux**

On écrit : $M = (\Sigma, E, \delta, I_0, F)$

Théorème de Kleene

À tout **AFND** M , correspond un **AFD** M' qui lui est **équivalent** (c'est-à-dire M et M' **acceptent le même langage**).

Idées de la preuve

- **Regroupement des états** : pour les transitions $(q, a, q_1), (q, a, q_2), \dots, (q, a, q_n)$, considérer l'ensemble des états d'arrivé $\{q_1, q_2, \dots, q_n\}$ comme un seul « grand état ».
- **ε -fermeture** : pour les transitions consécutives $(q, \varepsilon, q_1), (q_1, \varepsilon, q_2), \dots, (q_{n-1}, \varepsilon, q_n)$, où « q_n » est un état à partir duquel ne sorte aucune ε -transition, considérer les états de q à q_{n-1} comme équivalents à q_n .

Définition de l' ε -fermeture

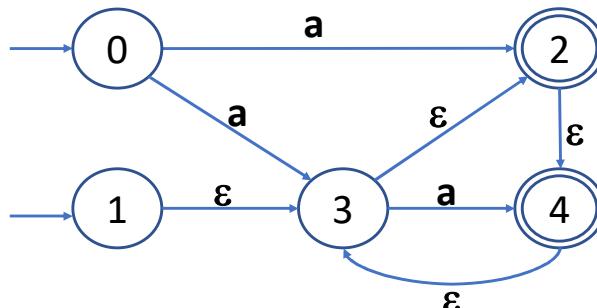
Soit $M = (\Sigma, E, \delta, I_0, F)$ un AFND et $X \subseteq E$.

L' ε – **fermeture de** X , noté $\widehat{\varepsilon}(X)$ est défini par :

$$\widehat{\varepsilon}(X) = X \bigcup \{q \in E \mid \text{il existe un chemin } \mathcal{C} \text{ d'un état } p \in X \text{ vers } q \text{ tel que } w(\mathcal{C}) = \varepsilon\}$$

- L' ε -fermeture d'un groupe d'états X , est l'ensemble des **états accessibles** depuis un état « p » de X en ne lisant que le mot vide « ε ».

L' ε -fermeture est une application de $\mathcal{P}(E)$ vers $\mathcal{P}(E)$.

Exemples

$$\widehat{\varepsilon}(\{0\}) = \{0\}$$

$$\widehat{\varepsilon}(\{1\}) = \{1, 2, 3, 4\} \quad \widehat{\varepsilon}(\{0, 1\}) = \{0, 1, 2, 3, 4\}$$

$$\widehat{\varepsilon}(\{2\}) = \{2, 3, 4\} \quad \widehat{\varepsilon}(\{1, 2\}) = \{1, 2, 3, 4\}$$

$$\widehat{\varepsilon}(\{3\}) = \{2, 3, 4\} \quad \widehat{\varepsilon}(\{2, 3, 4\}) = \{2, 3, 4\}$$

$$\widehat{\varepsilon}(\{4\}) = \{2, 3, 4\} \quad \widehat{\varepsilon}(\{0, 1, 2, 3, 4\}) = \{0, 1, 2, 3, 4\}$$

Propriétés de l' ε -fermeture

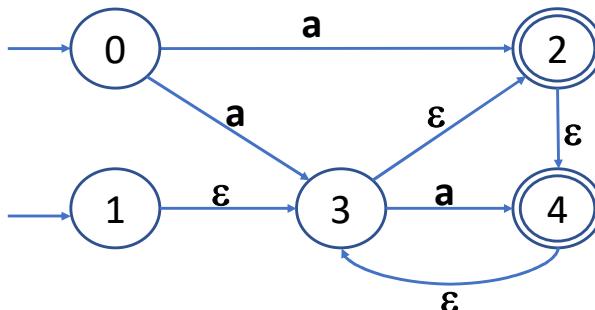
- $\widehat{\varepsilon}(\emptyset) = \emptyset$
- $X \subseteq \widehat{\varepsilon}(X)$
- $\widehat{\varepsilon}(X \cup Y) = \widehat{\varepsilon}(X) \cup \widehat{\varepsilon}(Y)$
- $\widehat{\varepsilon}(\widehat{\varepsilon}(X)) = \widehat{\varepsilon}(X)$

Algorithme de Kleene

- **Input** : $M = (\Sigma, E, \delta, I_0, F)$ un AFND.
- **Output** : $M' = (\Sigma', E', \delta', q'_0, F')$ un AFD équivalent à M .
(les éléments de E' seront des sous-ensembles de E).
- **Procédure** :
 - $\Sigma' = \Sigma$.
 - $q'_0 = \widehat{\varepsilon}(I_0)$.
 - Ajouter q'_0 à E' .

Algorithme de Kleene

- **Procédure** : (suite)
 - Pour chaque nouvel état p ajouté à E' faire :
 - Pour chaque symbole $a \in \Sigma$:
 - *) Calculer $\delta'(p, a) = \widehat{\varepsilon}(\bigcup_{e \in p} \delta(e, a))$;
 - *) Si $\delta'(p, a)$ est un nouvel état, l'ajouter à E' ;
 - Construire $F' = \{p \in E' \mid p \cap F \neq \emptyset\}$

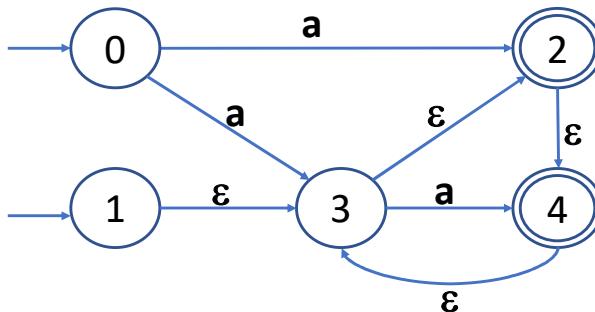
Exemple

$$\Sigma' = \{a\}$$

$$q_0 = \widehat{\varepsilon}(\{0, 1\}) = \{0, 1, 2, 3, 4\}$$

$$\delta'(q_0, a) = \widehat{\varepsilon}\left(\bigcup_{p \in \{0,1,2,3,4\}} \delta(p, a)\right) = \widehat{\varepsilon}(\delta(0, a) \cup \delta(3, a))$$

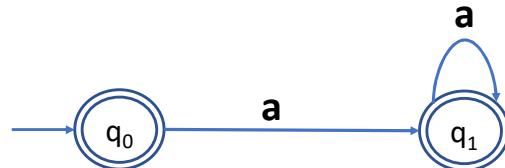
$$\delta'(q_0, a) = \widehat{\varepsilon}(\{2, 3, 4\}) = \{2, 3, 4\} = q_1$$

Exemple

$$\delta'(q_1, a) = \widehat{\varepsilon}\left(\bigcup_{p \in \{2,3,4\}} \delta(p, a)\right) = \widehat{\varepsilon}(\delta(3, a)) = \widehat{\varepsilon}(\{4\}) = \{2, 3, 4\} = q_1$$

$$F' = \{q_0, q_1\}$$

Exemple



$$\mathcal{L}(M) = (\varepsilon | a)a^* = a^*|a^+ = a^*$$

Théorème

Tout **langage régulier** est **accepté** par un **AFND** (et donc par l'**AFD** équivalent).

Algorithme de Thompson

- **Input** : une **expression régulière** « r » qui **dénote** le langage **régulier** « L ».
- **Output** : un **AFND** qui accepte « L ».

Automate généralisé normalisé

- Un **automate généralisé normalisé** est un **AFND** possédant les propriétés suivantes :
 - (1) il possède un **seul état initial** « i_0 » et un **seul état final** « f_0 »;
 - (2) aucune transition n'entre dans l'état « i_0 » et aucune transition ne sort de l'état final « f_0 ».
 - (3) à partir d'un état « e », il sort une **seule transition** avec un **symbole** de l'alphabet, ou bien **au plus deux ϵ -transitions**.

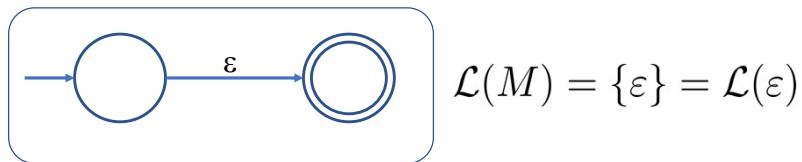
Algorithme de Thompson

► **Procédure :**

- Si $r = \emptyset$, construire :

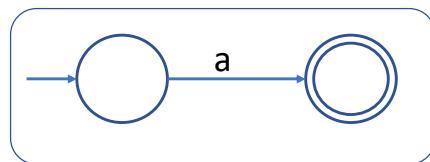


- Si $r = \epsilon$, construire :



Algorithme de Thompson

- Si $r = a \in \Sigma$, construire :



$$\mathcal{L}(M) = \{a\} = \mathcal{L}(a)$$

Soit M_s (resp. M_t) l'AFND acceptant $\mathcal{L}(s)$ (resp. $\mathcal{L}(t)$)



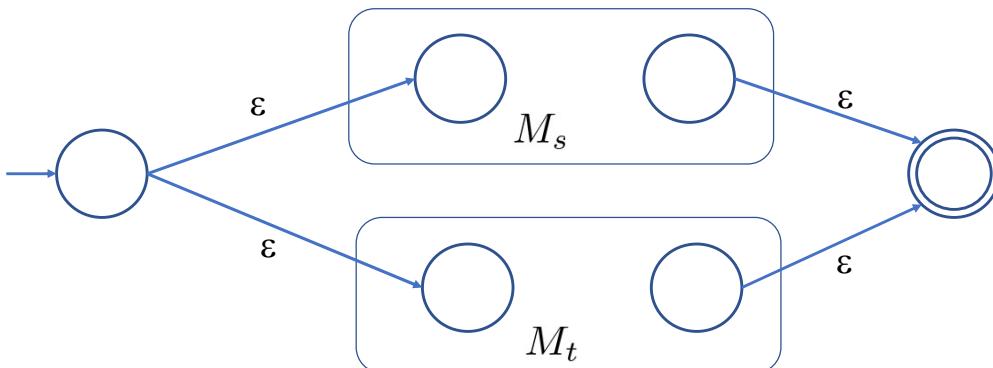
$$\mathcal{L}(M_s) = \mathcal{L}(s)$$



$$\mathcal{L}(M_t) = \mathcal{L}(t)$$

Algorithme de Thompson

- Si $r = s|t$, construire :



$$\mathcal{L}(M) = \mathcal{L}(M_s) \cup \mathcal{L}(M_t) = \mathcal{L}(s|t)$$

Algorithme de Thompson

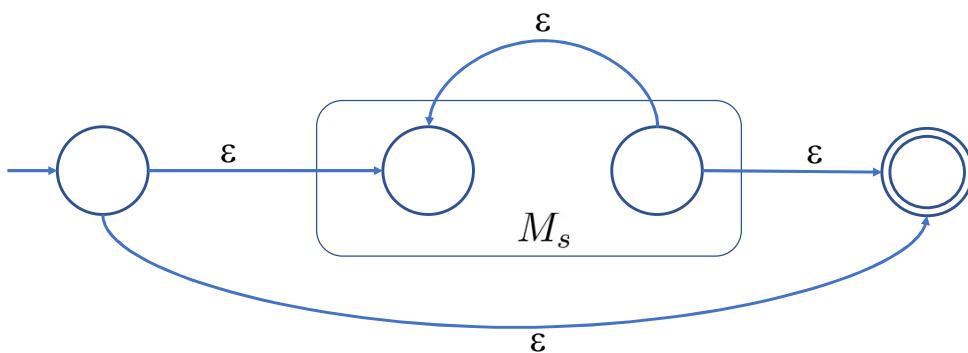
- Si $r = st$, construire :



$$\mathcal{L}(M) = \mathcal{L}(M_s)\mathcal{L}(M_t) = \mathcal{L}(st)$$

Algorithme de Thompson

- Si $r = s^*$, construire :

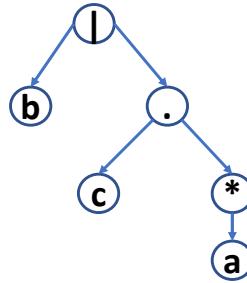


$$\mathcal{L}(M) = (\mathcal{L}(M_s))^* = \mathcal{L}(s^*)$$

Exemple

Construisons un AFND qui accepte $b|ca^*$

1. On construit un arbre binaire qui représente $b|ca^*$:



2. Parcourir l'arbre binaire en ordre postfixe : $bca * \cdot |$

Exemple

Décomposition de l'expression régulière :

$$r_1 = b$$

$$r_2 = c$$

$$r_3 = a$$

$$r_4 = r_3^*$$

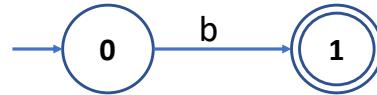
$$r_5 = r_2 \cdot r_4$$

$$r_6 = r_1 | r_5 = b | ca^*$$

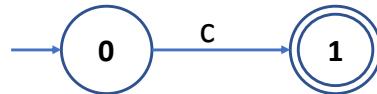
Exemple

3. Appliquer l'algorithme de Thompson

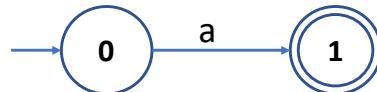
- Pour $r_1 = b$, on construit :



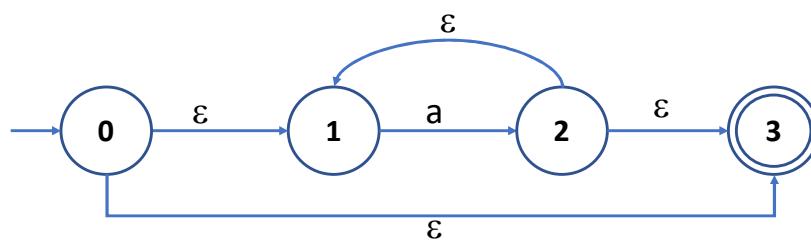
- Pour $r_2 = c$, on construit :

**Exemple**

- Pour $r_3 = a$, on construit :

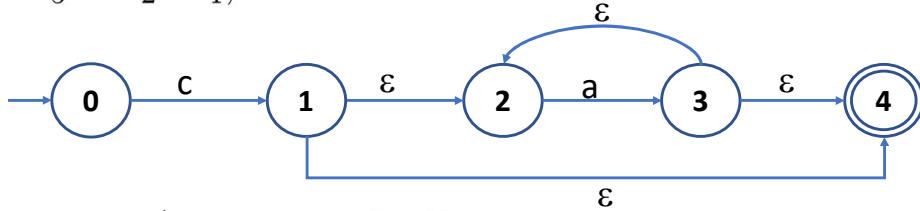


- Pour $r_4 = r_3^*$, on construit :

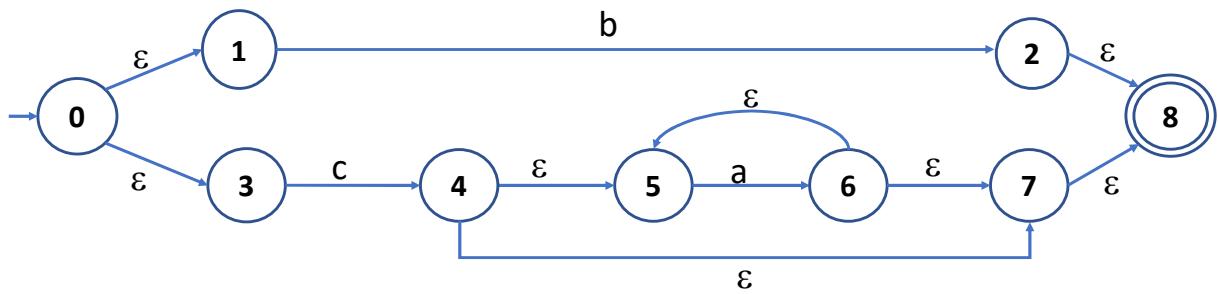


Exemple

- Pour $r_5 = r_2 \cdot r_4$, on construit :



- Pour $r_6 = r_1 | r_5$, on construit :



Théorème

Un **langage** est **régulier**, si et seulement s'il est accepté par un **AFD**.

Automate minimal

- L'algorithme de **déterminisation** d'un **AFND** produit souvent un **AFD** avec plus d'états que nécessaire.
- La **minimisation** d'un **AFD** vise à **réduire** au minimum le nombre des états de l'**AFD** tout en **préservant le langage accepté**.
- Il existe plusieurs algorithmes de minimisation dont le plus populaire est l'algorithme de **Hopcroft** de complexité **$O(n \log(n))$** où « n » est le nombre d'états de l'**AFD** de départ.
- L'algorithme de **minimisation** d'un **AFD** commence d'abord par une **simplification** de l'**AFD**.

Simplification d'un AFD

- **États accessibles** : un état d'un **AFD** qui peut être atteint depuis l'état initial q_0 .
- **États utiles** : un état d'un **AFD** qui est accessible et qui mène à un état final de l'automate.
- **Simplification** d'un **AFD** :
 - Supprimer les **états non accessibles** et toutes les **transitions** qui s'y rapportent.
 - Supprimer les **états non utiles** et toutes les **transitions** qui s'y rapportent.

Algorithm 1: Accessibles(M)

Input: M : AFD;
Output: A : Ensemble d'états;
Variables: P : Pile d'états; q : état;
begin

```

A := { $q_0$ };
empiler( $P, q_0$ );
while ( $P$  est non vide) do
|    $q$  := dépiler( $P$ );
|   for ( $a \in \Sigma$ ) et ( $\delta(q, a)$  existe) do
|   |   if ( $\delta(q, a) \notin A$ ) then
|   |   |   empiler( $P, \delta(q, a)$ );
|   |   |   A :=  $A \cup \{\delta(q, a)\}$ ;
|   |   end
|   end
|   end
retourner  $A$ ;
end

```

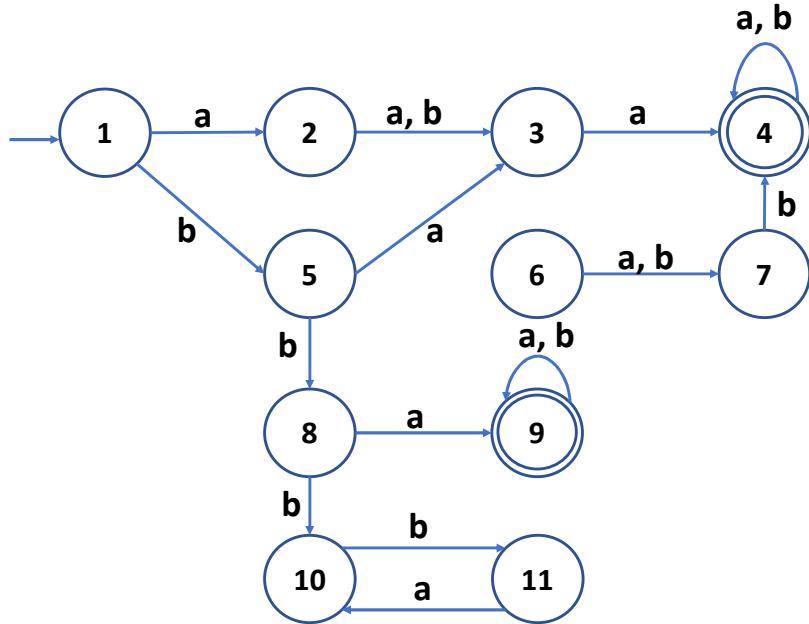
Algorithm 1: Utiles(M)

Input: M : AFD;
Output: A, B, U : Ensembles d'états;
Variables: P : Pile d'états; q : état;
begin

```

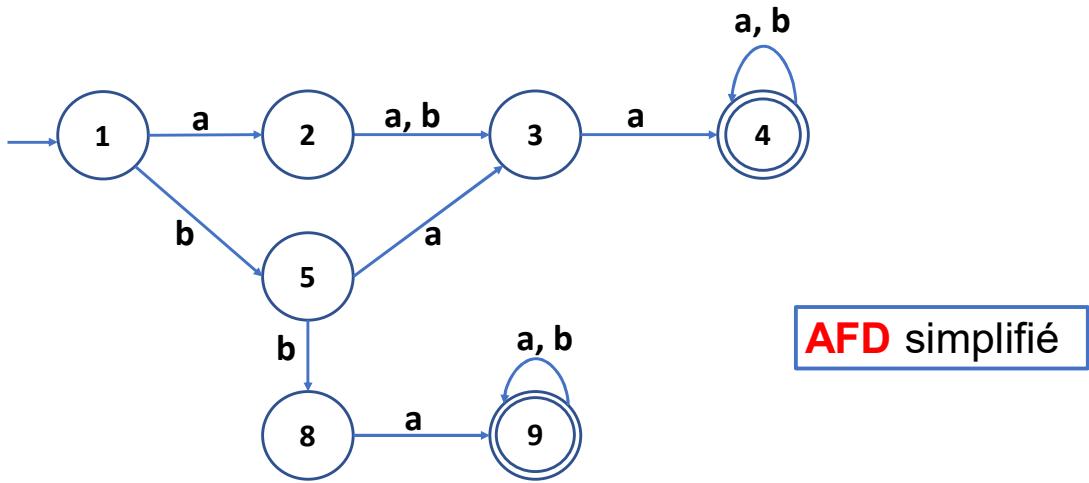
A := Accessibles( $M$ );    U :=  $\emptyset$ ;
for ( $p \in A$ ) do
|   if ( $p \in F$ ) then
|   |   U :=  $U \cup \{p\}$ ;
|   else
|   |   B :=  $\{p\}$ ;    P := pile-vide();    empiler( $P, p$ );
|   |   while ( $P$  est non vide) et ( $p \notin U$ ) do
|   |   |   q := dépiler( $P$ );
|   |   |   for ( $a \in \Sigma$ ) et ( $\delta(q, a)$  existe) do
|   |   |   |   if ( $\delta(q, a) \in F$ ) then
|   |   |   |   |   U :=  $U \cup \{\delta(q, a)\}$ ;
|   |   |   |   else
|   |   |   |   |   if ( $\delta(q, a) \notin B$ ) then
|   |   |   |   |   |   empiler( $P, \delta(q, a)$ );
|   |   |   |   |   B :=  $B \cup \{\delta(q, a)\}$ ;
|   |   |   |   end
|   |   |   end
|   |   end
|   end
|   retourner  $U$ ;
end

```

Exemple**Exemple**

- **États accessibles** : 1, 2, 3, 4, 5, 8, 9, 10, 11.
 - On supprime les états 6 et 7 et toutes les **transitions** qui s'y rapportent.
- **États utiles** : 1, 2, 3, 4, 5, 8, 9.
 - On supprime les états 10 et 11 et toutes les **transitions** qui s'y rapportent.

Exemple



Algorithme de Hopcroft

- ▶ **Input** : $M = (\Sigma, E, \delta, q_0, F)$ un AFD simplifié.
- ▶ **Output** : $M' = (\Sigma, E', \delta', q'_0, F')$ un AFD minimal équivalent à M .
(les éléments de E' seront des sous-ensembles de E).
- ▶ **Idée de l'algorithme** : regrouper les états équivalents.

Algorithme de Hopcroft

- **Équivalence de Néode** : deux états « p » et « q » sont équivalents si et seulement si l'on reconnaît le même langage en prenant comme états initiaux « p » et « q ».

Soient M un AFD simplifié et p un état de M .

On note \mathcal{L}_p le langage des mots sur Σ acceptés par M lorsque l'état initial est p . Formellement :

$$\mathcal{L}_p = \{x \in \Sigma^* \mid \lambda = (p, \dots, q) \text{ est un chemin dans } M, w(\lambda) = x \text{ et } q \in F\}$$

Algorithme de Hopcroft

L'équivalence de Néode est la relation binaire définie sur E par :

$$p \sim q \iff \mathcal{L}_p = \mathcal{L}_q$$

Il s'agit bien d'une **relation d'équivalence**.

L'**ensemble quotient** E / \sim de cette relation d'équivalence est l'ensemble des états de l'**automate minimal** équivalent à M .

En effet, les états appartenant à une **même classe d'équivalence** reconnaissent le même langage : on doit alors les **fusionner**.

Algorithme de Hopcroft

Première remarque : si $p \in F$ et $q \notin F$, alors $p \not\sim q$.

En effet, $\varepsilon \in \mathcal{L}_p$ (car $p \in F$) et $\varepsilon \notin \mathcal{L}_q$ (car $q \notin F$).

L'algorithme de Hopcroft procède par **partitionnement** des états :

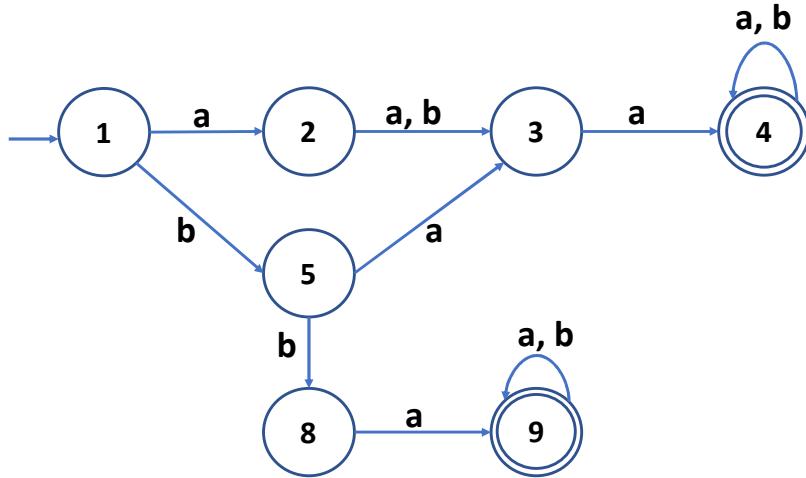
- La première partition de l'ensemble des états E donne deux classes : F et $E \setminus F$.
- Si une classe C d'une partition contient deux états p et q et s'il existe un symbole $a \in \Sigma$ tel que $\delta(p, a)$ et $\delta(q, a)$ ne mène pas à la même classe, il faut partager la classe C en deux sous-classes : une contenant p et l'autre contenant q .

Algorithm 1: Minimisation(M)

```

Input:  $M$  : AFD; /* simplifié */
Output:  $N$  : AFD; /* AFD minimal équivalent à  $M$  */
begin
     $P_0 := (F, E \setminus F);$ 
     $i := 0;$ 
    for ( $C \in P_i$ ) do
         $i := i + 1;$ 
        for ( $a \in \Sigma$ ) do
            for ( $(p, q) \in C^2$ ) do
                if ( $(\delta(p, a)$  et  $\delta(q, a)$  ne conduisent pas vers la même
                     classe dans  $P_{i-1}$ ) then
                    | Partager  $C$  en deux sous-classes dans  $P_i$ ;
                end
            end
        end
    end
    Les états de  $N$  sont les classes de la dernière partition trouvée;
end

```

Exemple**Exemple**

$P_0 = (\mathcal{C}_1, \mathcal{C}_2)$, avec $\mathcal{C}_1 = \{1, 2, 3, 5, 8\}$ et $\mathcal{C}_2 = \{4, 9\}$

Classes	États	a	b
\mathcal{C}_1	1	2	5
	2	3	3
	3	4	-
	5	3	8
	8	9	-
\mathcal{C}_2	4	4	4
	9	9	9

Classes	États	a	b
\mathcal{C}_1	1	\mathcal{C}_1	\mathcal{C}_1
	2	\mathcal{C}_1	\mathcal{C}_1
	3	\mathcal{C}_2	-
	5	\mathcal{C}_1	\mathcal{C}_1
\mathcal{C}_2	8	\mathcal{C}_2	-
	4	\mathcal{C}_2	\mathcal{C}_2
	9	\mathcal{C}_2	\mathcal{C}_2

Exemple

$P_1 = (\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3)$, avec $\mathcal{C}_1 = \{1, 2, 5\}$, $\mathcal{C}_2 = \{3, 8\}$ et $\mathcal{C}_3 = \{4, 9\}$

Classes	États	a	b
\mathcal{C}_1	1	\mathcal{C}_1	\mathcal{C}_1
	2	\mathcal{C}_2	\mathcal{C}_2
	5	\mathcal{C}_2	\mathcal{C}_2
\mathcal{C}_2	3	\mathcal{C}_3	-
	8	\mathcal{C}_3	-
\mathcal{C}_3	4	\mathcal{C}_3	\mathcal{C}_3
	9	\mathcal{C}_3	\mathcal{C}_3

Exemple

$P_2 = (\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4)$, avec $\mathcal{C}_1 = \{1\}$, $\mathcal{C}_2 = \{2, 5\}$, $\mathcal{C}_3 = \{3, 8\}$

et $\mathcal{C}_4 = \{4, 9\}$

Classes	États	a	b
\mathcal{C}_1	1	\mathcal{C}_2	\mathcal{C}_2
\mathcal{C}_2	2	\mathcal{C}_3	\mathcal{C}_3
	5	\mathcal{C}_3	\mathcal{C}_3
\mathcal{C}_3	3	\mathcal{C}_4	-
	8	\mathcal{C}_4	-
\mathcal{C}_4	4	\mathcal{C}_4	\mathcal{C}_4
	9	\mathcal{C}_4	\mathcal{C}_4

L'AFD minimal contient 4 états.

Construction de l'AFD minimal

- On transforme chaque **classe** en état dans l'**AFD minimal**.
- L'état initial de l'**AFD minimal** est la classe qui contient l'état initial q_0 de l'AFD de départ.
- Les états finaux de l'**AFD minimal** sont les classes qui ne contiennent que des états finaux de l'AFD de départ.
- Les transitions pour une classe sont fusionnées en une seule transition de l'**AFD minimal**.

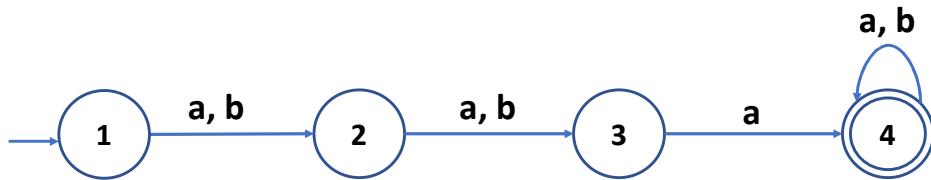
Exemple

L'**AFD minimal** de notre exemple est :

- $E = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$.
- $q_0 = \mathcal{C}_1$.
- $F = \mathcal{C}_4$.

δ	a	b
\mathcal{C}_1	\mathcal{C}_2	\mathcal{C}_2
\mathcal{C}_2	\mathcal{C}_3	\mathcal{C}_3
\mathcal{C}_3	\mathcal{C}_4	-
\mathcal{C}_4	\mathcal{C}_4	\mathcal{C}_4

Exemple



$$\mathcal{L}(M) = (a|b)(a|b)a(a|b)^*$$

$\mathcal{L}(M)$ est le langage des mots dont le troisième caractère à partir de la gauche est un *a*.

Autres propriétés de clôture pour les langages réguliers

- La classe des langages **réguliers** est **close** par :
 - **Intersection**.
 - **Miroir** (voir TD).
 - **Complémentaire**.
 - **Différence**.
 - **Différence symétrique**.
 - **Homomorphisme** (voir TD).

Lemme de pompage

Soit L un langage sur un alphabet Σ .

Méthodes pour prouver que L est **régulier** :

- Trouver une **expression régulière** qui dénote L .
- Trouver un **AFD** ou un **AFND** qui accepte L .
- Utiliser les **propriétés de clôture** pour les langages réguliers.

Lemme de pompage

Pour prouver que L n'est pas **régulier**, on utilise le **lemme de pompage**.

Énoncé du Lemme de pompage

Si L est **régulier**, alors :

$(\exists N \in \mathbb{N}) (\forall w \in L) : |w| \geq N \implies w = xyz$ avec

- (i) $|xy| \leq N$
- (ii) $|y| \geq 1$ ($y \neq \varepsilon$)
- (iii) $(\forall k \in \mathbb{N}) : xy^k z \in L$

Preuve du Lemme de pompage

Soit L un langage **régulier**.

Il existe un **AFD minimal** M qui accepte L .

Soit N le nombre des états de l'AFD M . Il y a 2 cas :

- L est **fini** : $|w| < N$, pour tout mot $w \in L$. Donc le Lemme est vérifié, car aucun mot de L n'a une longueur $\geq N$.

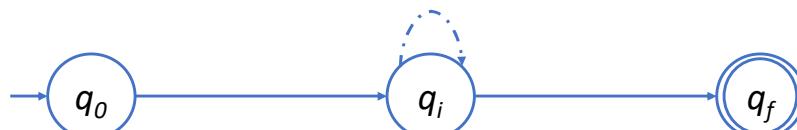
- L est **infini** : L contient au moins un mot w de longueur $\geq N$.

Donc, le **chemin de reconnaissance** de w contient un **circuit**.

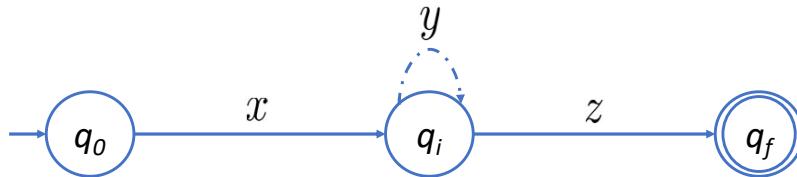
Preuve du Lemme de pompage

Le **chemin de reconnaissance** de w contient un **circuit** :

- il part de l'état initial q_0 ;
- il arrive à un état q_i autour duquel il va former un circuit;
- ensuite, il va partir de q_i pour atteindre un état final q_f .



Preuve du Lemme de pompage



- De q_0 à q_i , M va lire un mot x ;
- Le circuit autour de q_i nécessite la lecture d'un mot y ;
- De q_i à q_f , M va lire un mot z .

Donc, w se décompose sous la forme xyz .

Preuve du Lemme de pompage

Le chemin entre q_0 et q_i étant élémentaire, donc $|xy| \leq N$.

Pour avoir un circuit, il faut que $y \neq \varepsilon$ (i.e., $|y| \geq 1$).

Le mot $xy^0z = xz \in L$, car relie q_0 à un état final $q_f \in F$.

On peut reboucler sur l'état q_i autant de fois que l'on veut.

Par suite, le mot $xy^kz \in L$, pour tout entier k .

Application du Lemme de pompage

Les langages suivants ne sont pas **réguliers** :

$$\{a^n b^n \mid n \geq 0\}$$

$$\{w \in \{a, b\}^* \mid w \text{ est un palindrome}\} \text{ (voir TD)}$$

$$\{a^{n^2} \mid n \geq 0\} \text{ (voir TD)}$$

$$\{a^n \mid n \text{ premier}\} \text{ (voir TD)}$$

Application du Lemme de pompage

$$L = \{a^n b^n \mid n \geq 0\}$$

Si L est **régulier**, il vérifie le **lemme de pompage**.

$(\exists N \in \mathbb{N}) (\forall w \in L) : |w| \geq N \implies w = xyz$ avec

$$(i) |xy| \leq N$$

$$(ii) |y| \geq 1 (y \neq \varepsilon)$$

$$(iii) (\forall k \in \mathbb{N}) : xy^k z \in L$$

Prenons le mot $w = a^N b^N$. On a : $w \in L$ et $|w| = 2N \geq N$.

Application du Lemme de pompage

Donc $w = xyz$ avec (i), (ii) et (iii).

Comme $|xy| \leq N$, alors xy est un préfixe de a^N .

Par suite, $x = a^i$, $y = a^j$ et $z = a^k b^N$, avec $i + j + k = N$ et $j \geq 1$, car $y \neq \varepsilon$ (d'après (ii)).

Alors, $xy^0 z = a^i a^k b^N = a^{i+k} b^N$, avec $i + k = N - j \neq N$, car $j \geq 1$.

Donc $xy^0 z \notin L$: **Contradiction** avec (iii).

Conclusion : L n'est pas régulier.

Remarque

La **réciproque** du lemme de pompage est fausse.



Outil Flex



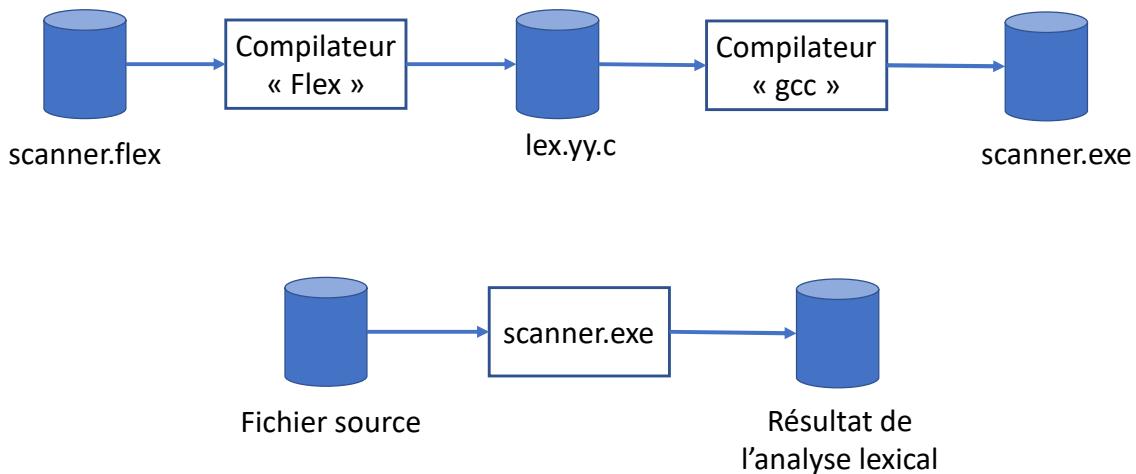
Description de Flex

- **Flex (Fast Lexer)** est un programme de **génération d'analyseurs lexicaux**.
- Un fichier **Flex** (format texte) contient la **description** d'un **analyseur lexical à générer**.
- Cette description est présentée sous forme de paires d'**expressions régulières** et du **code C** (ou C++).
- **Flex** produit comme résultat un fichier contenant le code C du future **analyseur lexical**.
- Ce fichier est nommé « **lex.yy.c** ».

Description de Flex

- La **compilation C** du fichier « **lex.yy.c** » génère le **code exécutable de l'analyseur lexical**.
- Lorsque l'exécutable est appelé, il analyse le fichier source pour chercher les **occurrences d'expressions régulières**.
- Lorsqu'une chaîne de l'entrée correspond à une **expression régulière**, **l'analyseur lexique** exécute le code C correspondant.
- Un fichier **Flex** est un fichier texte dont l'extension est « **.flex** » (ou « **.lex** » ou même « **.l** »).

Description de Flex



Description de Flex

- La **chaîne** des opérations exécutées par **Flex** :
 - Analyse syntaxique de chaque **expression régulième** « r_i ».
 - Conversion de « r_i » en un **AFND** (algorithme de **Thompson**).
 - Conversion de l'**AFND** en **AFD** (algorithme de **Kleene**).
 - Minimisation de l'**AFD** (algorithme de **Hopcroft**).
 - Conversion de l'**AFD minimal** en un programme C/C++ et ajout du code C/C++ (**lex.yy.c**).

Premier exemple

```
%%
"a" printf("z");
"z" printf("a");
%%
int main()
{
    yylex();
    return 0;
}
int yywrap()
{
    return 1;
}
```

Fichier : scan.l

1) **Compilation** avec Flex :

>> flex scan.l

Elle produit le fichier **lex.yy.c**.

2) **Compilation** avec gcc :

>> gcc lex.yy.c -o scan.exe

Elle produit le fichier exécutable.

Premier exemple

```
c:\TP compilation>scan
je vais aller au zoo demain.
je vzis zller zu aoo demzin.
^C
c:\TP compilation>scan < source.txt
je vzis zller zu aoo demzin.^C
c:\TP compilation>
```

Exécution de l'analyseur lexical

Premier exemple

- Examinons le fichier « **scan.l** » :
 - Les **règles** de l'**analyseur lexical** sont écrites entre les symboles « **%%** ».
 - Une **règle** est une paire « **expression régulière / action** » :
 - « **"a" printf("z");** » est une règle :
 - L'**expression régulière** est « **"a"** », c'est-à-dire une lettre « **a** ».
 - L'**action** est « **printf("z");** » (c'est du code C/C++).
 - **Signification** : remplacer toute occurrence de « **a** » dans le texte d'entrée par une occurrence de « **Z** ».

Premier exemple

- **Remarques :**

- Les symboles « `%%` » et les **expressions régulières** doivent commencer en **première colonne** d'une **nouvelle ligne**.
- Il faut laisser **au moins un espace** (ou **tabulation**) entre l'**expression régulière** et l'**action** correspondante.
- Si le code C/C++ contient plusieurs instructions, il faut les mettre en accolades « `{}` » et « `}` ».
- La fonction « `yylex` » effectue un appel explicite de l'analyseur lexical.

Deuxième exemple

Un analyseur lexical qui compte le nombre de lignes et de caractères de l'entrée.

```
%{
    int rows = 0, cars = 0;
}%
%
\n  rows++;
.  cars++;
%%
int main()
{
    yylex();
    printf("il y a %d lignes\n", rows);
    printf("et %d characters", cars);
    return 0;
}
int yywrap()
{
    return 1;
}
```

Deuxième exemple

- La **section** entre les symboles « `%{` » et « `%}` » est réservée pour la déclaration (en langage C/C++) des **variables** et des **fonctions globales**.
- Les symboles « `%{` » et « `%}` » doivent être mis en **première colonne** d'une **nouvelle ligne**.
- Les variables déclarées dans cette section sont accessibles par les fonctions « `main` », « `yylex` » et « `yywrap` ».
- L'**expression régulière** « `\n` » correspond à un caractère « **retour chariot** » unique et l'**expression régulière** « `.` » dénote **n'importe quel caractère** (un seul caractère) autre que le caractère « **retour chariot** ».

Deuxième exemple

- Lorsque l'analyseur lexical est terminée, les instructions après l'appel « `yylex` » seront exécutées.

```
source.txt - ...
Fichier Edition Format Affichage
Aide
premiere ligne
deuxieme ligne
troisieme ligne
|
< >
Windows (CRLF) UTF-8

c:\TP compilation>scan2 < source.txt
il y a 3 lignes
et 43 caracteres
```

Format et contenu d'un fichier Flex

- Le code **Flex** comprend **quatre sections** :
 - La section des **déclarations globales**.
 - La section des **définitions régulières**.
 - La section des **règles**.
 - La section du **code auxillaire**.

Format et contenu d'un fichier Flex

```
%{
    /* Section des déclarations globales */
}%
    /* Section des définitions régulières */
%%
    /* Section des règles */
%%
    /* Section du code auxillaire */
```

Section des déclarations globales

- Elle contient les **déclarations** (en C/C++) des :
 - **variables globales**.
 - **types**.
 - **fonctions globales** (en C/C++).
- Cette section doit être écrite entre les symboles « **%{** » et « **%}** ».

Section des définitions régulières

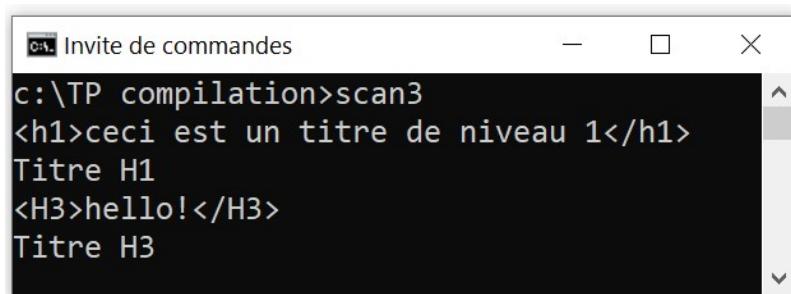
- Elle contient les **définitions régulières** (pour nommer les expressions régulières).
- Une **définition régulière** comporte un **identificateur** (comme en C/C++), suivi d'au moins un **caractère blanc** (espace ou tabulation), suivie d'une **expression régulière**.
- Pour référencer une **expression régulière** déjà nommée, il suffit d'écrire son **identificateur** entre deux accolades « **{** » et « **}** »).
- Une **définition régulière** doit tenir sur une **seule ligne**.

Troisième exemple

Un analyseur lexical qui détecte les balises html de titres.

```
level [1-6]
title "<(h|H){level}>.*?"</(h|H){level}>""
%%
{title} printf("Titre H%d", yytext[2] - '0');
%%
int main()
{
    yylex();
    return 0;
}
int yywrap()
{
    return 1;
}
```

Troisième exemple



The screenshot shows a Windows Command Prompt window with the title "Invite de commandes". The command entered is "c:\TP compilation>scan3". The output displays the tokens extracted from the input string: "<h1>ceci est un titre de niveau 1</h1>", "Titre H1", "<H3>hello!</H3>", and "Titre H3".

```
c:\TP compilation>scan3
<h1>ceci est un titre de niveau 1</h1>
Titre H1
<H3>hello!</H3>
Titre H3
```

Section des règles

- Elle contient les **règles** (**expression régulière / action**).
- Elle vient après le premier symbole « **%%** » (qui est obligatoire).
- Une **action** doit être écrite dans la **même ligne** que son **expression régulière**.
- **Flex** utilise les **expressions régulières étendues** (globalement standards, mais elles présentent quelques spécificités et différences par rapport aux **regex** utilisées dans d'autres langages ou outils comme **Python**, **JavaScript**, ou les outils **Unix (grep)**).
- L'**analyseur lexical** déclenche une **action** chaque fois qu'il trouve une partie du texte qui correspond à son **expression régulière**.

Section du code auxiliaire

- Le contenu de cette section sera copiée tel quel dans le fichier « **lex.yy.c** ».
- Elle contient les routines complémentaires (fonctions C/C++) qui peuvent être appelées par l'**analyseur lexical**.
- Elle contient également les deux fonctions « **main** » et « **yywrap** ».
- La fonction « **yywrap** » gère la fin de l'entrée dans les analyseurs lexicaux générés par **Flex**. Elle permet :
 - d'arrêter l'analyse à la fin du fichier (comportement par défaut).
 - de passer à un nouveau fichier pour continuer l'analyse.
 - d'implémenter des comportements personnalisés pour gérer des flux d'entrée complexes.

Quatrième exemple

Un analyseur lexical qui calcule $n!$.

```
%{
#include<stdlib.h>
int fact(int n);
%}
nb [0-9] +
%%
{nb}"!" { printf("%d", fact(atoi(yytext))); }
%
int fact(int n)
{ return (n == 0 ? 1 : n * fact(n - 1)); }
int main()
{
    yylex();
    return 0;
}
int yywrap()
{
    return 1;
}
```

Expression régulière	Signification
x	le caractère x
.	n'importe quel caractère, sauf \n
[xyz]	x, y ou z
[adi-kp]	a, d, i, j, k ou p
[^a-z]	tous les caractères sauf a, b, ..., z
r*	0, 1 ou plusieurs occurrences de r
r+	1 ou plusieurs occurrences de r
r?	0 ou une occurrence de r
r{2,5}	2, 3, 4 ou 5 occurrences de r
r{2,}	2 ou plusieurs occurrences de r
r{3}	exactement 3 occurrences de r

Expression régulière	Signification
{nom}	expansion de la définition nom
\x	caractère d'échapement (\n, \t, \0, ...)
\123	caractère de code ASCII 123
\x2A	caractère de code hexadécimal x2A
(r)	une occurrence de r (équivalent à r)
r s	une occurrence de r ou de s
rs	une occurrence de r, suivie d'une occurrence de s
r/s	une occurrence de r, sauf si elle est suivie d'une occurrence de s
^r	une occurrence de r, mais uniquement au début d'une ligne
r\$	une occurrence de r, mais uniquement à la fin d'une ligne
<< EOF >>	le caractère de fin de fichier

Fonctionnement de l'analyseur lexical généré par Flex

- Lorsque l'**analyseur lexical** est exécuté, il analyse l'entrée pour rechercher les chaînes qui correspondent à l'une de ses **expressions régulières**.
- L'**analyseur lexical** applique l'un de ces trois principes suivants :
 - Principe de la **plus longue préfixe**.
 - Principe de la **première règle**.
 - Principe de la **règle par défaut**.

Fonctionnement de l'analyseur lexical généré par Flex

- **Principe de la plus longue préfixe :**

- si un préfixe de longueur « k » d'une chaîne « CH » est reconnu par l'**expression régulière** d'une règle « r1 »;
- et si un préfixe de longueur « m » de chaîne « CH » est reconnu par une autre **expression régulière** d'une autre règle « r2 »;
- alors l'analyseur applique la règle qui correspond à « max(k, m) ».

Fonctionnement de l'analyseur lexical généré par Flex

Exemple :

```
si      printf("KeyWord : si");
[a-z]* printf("ID : %s", yytext);
```

- Si l'entrée est « six » :
 - Le préfixe « si » est reconnu par l'**E.R** de la première règle.
 - Mais, le préfixe « six » est reconnu par l'**E.R** de la deuxième règle.
 - L'analyseur appliquera la seconde règle : « six » est un ID.

Fonctionnement de l'analyseur lexical généré par Flex

- **Principe de la première règle :**

- si un préfixe de même longueur d'une chaîne « CH » est reconnu par plusieurs **expressions régulières**;
- alors l'analyseur applique la première règle.

Fonctionnement de l'analyseur lexical généré par Flex

Exemple :

```
si      printf("KeyWord : si");
[a-z]* printf("ID : %s", yytext);
```

- Si l'entrée est « si » :

- Le préfixe « si » est reconnu par les deux **E.R** des deux règles.
- L'analyseur appliquera la première règle : « si » est un KeyWord.

Fonctionnement de l'analyseur lexical généré par Flex

- **Principe de la règle par défaut :**
 - si aucune correspondance n'est trouvée, alors l'**analyseur** active la **règle par défaut** qui consiste à **recopier le texte tel quel** dans la sortie standard.
- **La variable globale « yytext » :**
 - une fois un texte est reconnu, l'**analyseur** le **stocke** dans la **variable « yytext »** (de type « **char *** »).
 - sa longueur est également stockée dans la variable « **yyleng** » (de type « **int** »).
 - ensuite, l'**action de la règle choisie** est **exécutée** et l'**analyseur continue** son travail.

Les actions

- **Action d'une règle** : une suite d'instructions en C/C++.
- **Actions spéciales** :

- **Action vide** : permet de **effacer** un texte.

Exemple :

```
%%
bonjour /* action vide */
```

- Cette règle permet d'effacer toute occurrence du mot « bonjour ».

Les actions

- **Action ECHO** : permet de **copier le contenu** de « **yytext** » (équivalent à « **printf("%s", yytext);** »).

```
%%
bonjour ECHO;
```

- **Action |** : permet de **partager une action** par un ensemble d'**expressions régulières**.

```
%%
a  |
ab |
abc ECHO;
```

Les actions

- **Action yymore()** : demande à l'**analyseur lexical** de **concaténer** le contenu courant de « **yytext** » à son contenu précédent (au lieu de le remplacer).

```
%%
java ECHO; yymore();
script ECHO;
```

- Si l'entrée est « **javascript** », l'**analyseur** va reconnaître en premier lieu le préfixe « **java** » et le **recopier** dans « **yytext** ».
- Ensuite, l'**analyseur** reconnaîtra le suffixe « **script** ».
- Comme il y avait appel à « **yymore()** », l'**analyseur** concatène « **script** » à « **java** » : le contenu de « **yytext** » est alors « **javascript** ».

Les actions

- **Action `yyless()`** : demande à l'**analyseur lexical** de **reculer** « n » caractères sur le **lexème récemment détecté**.

```
%%
abracadabra ECHO; yyless(3);
[a-z]*      ECHO;
```

- Si l'entrée est « **abracadabra** », l'**analyseur** va reconnaître en premier lieu toute la chaîne « **abracadabra** », puis il va reculer de « **3** » caractères : le prochain entrée à analyser est « **bra** ».
- Le résultat affiché est donc « **abracadabrabra** ».

Les actions

- **Action `yyterminate()`** : déclenche l'**arrêt** de l'**analyseur lexical** en retournant la valeur « **0** » indiquant que tout est fait.
- Les variables globales « **`yyin`** » et « **`yyout`** » :
 - « **`yyin`** » permet de spécifier le **fichier d'entrée** à analyser (qui est par défaut, l'**entrée standard**).
 - « **`yyout`** » permet de spécifier le **fichier de sortie** dans lequel l'**analyseur** va **écrire le résultat** de l'**analyse lexicale** (qui est par défaut, la **sortie standard**).

Cinquième exemple

Un analyseur lexical du fragment du langage ATLAS.

```
%{
#include<stdlib.h>
%
nb [0-9] +
id [a-z][a-z0-9]*
op "<="|"<"|">="|">"|"="|"<>"
kw alors|finsi|si|sinon
%%
{kw} printf("keyword: %s\n", yytext);
{id} printf("id: %s\n", yytext);
{nb} printf("nb: %d\n", atoi(yytext));
{op} printf("oprel: %s\n", yytext);
"(" |
")" printf("parenthese: %c\n", yytext[0]);
[ \t\n] ;
.   { printf("error: %c illegal\n", yytext[0]); exit(-1); }
%%
int main()
{ yylex(); return 0; }
int yywrap()
{ return 1; }
```

Cinquième exemple

```
source.txt - Blo... — □ ×
Fichier Edition Format Affichage Aide
si(temp > 38)
    alors
        fievre
    sinon
        normal
finsi
< >
100% Windows (CRLF) UTF-8
```

```
C:\TP compilation>scan4 < source.txt
keyword: si
parenthese: (
id: temp
oprel: >
nb: 38
parenthese: )
keyword: alors
id: fievre
keyword: sinon
id: normal
keyword: finsi
```



TP en classe : Analyseur lexical du langage ATLAS



TP N° 1

- Écrivez en **Flex** l'**analyseur lexical** complet du langage **ATLAS**.