Analyse Lexicale

Théorie des langages et de compilation

R. EZZAHIR[1]

[1]Département d'Informatique
Ecole Nationale des Sciences Apliquées
ENSA-Agadir
Université Ibn Zohr

4ème année Génie Informatique, 2013/2014

Sommaire

Pourquoi faire une analyse lexicale ?

- Simplifier considérablement l'analyse.
  - Éliminer les espaces.
  - Éliminer les commentaires.
  - Convertir les données au début.
- Convertir la description physique d'un programme en une séquence de jetons.
  - Chaque jeton est associé à un lexème.
  - Chaque jeton peut avoir des attributs optionnels.
  - Le flux de jetons sera utilisé dans l'analyseur pour récupérer la structure du programme.

Analyse lexicale

Convertir la description physique d'un programme en une séquence de jetons.

```
While(ip<z)
  ++ip;
```

Analyse lexicale

Convertir la description physique d'un programme en une séquence de jetons.

```
While(ip<z)
    ++ip;
```

w	h	i	l	e		(	i		<		z	\	n	\	t	+	+	i	p	;

Analyse lexicale

Convertir la description physique d'un programme en une séquence de jetons.

```
While(ip<z)
  ++ip;
```

w	h	i	l	e		(	i		<		z	\	n	\	t	+	+	i	p	;

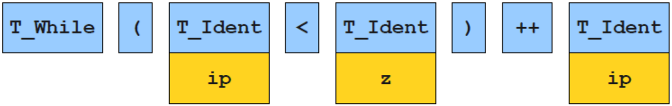
T_While	(	T_Ident	<	T_Ident	)	++	T_Ident
		ip		z			ip

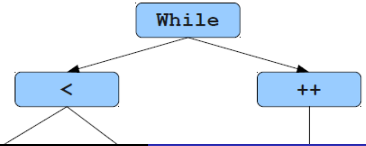
Analyse lexicale

Convertir la description physique d'un programme en une séquence de jetons.

```
While(ip<z)
  ++ip;
```

w	h	i	l	e		(	i		<		z	\	n	\	t	+	+	i	p	;





Défis de l'analyse lexicale

Défis de l'analyse lexicale

- Comment faire pour découper le programme en lexèmes ?
- Comment étiqueter chaque lexème correctement ?

FORTRAN : Les espaces ne sont pas pertinents

DO 5 I = 1,25
DO5I = 1,25
Il peut être difficile de dire où découper l'entrée.

Les mots clés peuvent être utilisés comme identificateurs

IF THEN THEN THEN = ELSE; ELSE ELSE = IF
Il peut être difficile de déterminer la façon d'étiqueter des lexèmes.

Définition d'une analyse lexicale

- Définir un ensemble de jetons.
- Définir l'ensemble des lexèmes associé à chaque jeton.
- Définir un algorithme pour résoudre les conflits qui se posent entre lexèmes.

Problématique

- But de la théorie des langages
- Un modèle d'aide
- Problèmatique

Le français est un langage, Java également. Le but de la théorie des langages et de donner un modèle de ce qu'est un langage.

Problématique

- But de la théorie des langages
- Un modèle d'aide
- Problèmatique

- pour pouvoir décrire un langage ;
- pour fabriquer une machine capable de reconnaître les textes qui appartiennent à un langage donné.

Problématique

- But de la théorie des langages
- Un modèle d'aide
- Problèmatique

il faut donner une description finie d'un objet en général infini : il y a en effet une infinité de textes français, une infinité de programmes Java

Alphabets, Mots, et Langages

Alphabet

Un alphabet Σ (ou vocbulaire, ou lexique) est un ensemble fini de symboles. On note Σ^* l'ensemble des mots sur Σ.

Mot

Un mot (ou phrase) sur un vocabulaire Σ est une séquence finie d'éléments de Σ.

Alphabets

1. $\Sigma=\{a, b\} \Rightarrow \Sigma^* =\{a,b, ab, \ldots, aaaa, aabb, \ldots\}$
2. {if, then, else, begin, end, :=, ;,(,), A, B, C, 1,2}

Mots

1. aaaa, aabb
2. if A then B := 1 else C := 2
   if if if A begin

Alphabets, Mots, et Langages

Alphabet

Un alphabet Σ (ou vocbulaire, ou lexique) est un ensemble fini de symboles. On note Σ^* l'ensemble des mots sur Σ.

Alphabets

1. Σ={a, b} $\Rightarrow$ Σ^* ={a,b, ab,..., aaaa, aabb, ...}
2. {if, then, else, begin, end, :=, ;,(,), A, B, C, 1,2}

Mot

Un mot (ou phrase) sur un vocabulaire Σ est une séquence finie d'éléments de Σ.

Mots

1. aaaa, aabb
2. if A then B := 1 else C := 2
   if if if A begin

Alphabets, Mots, et Langages

Alphabet

Un alphabet Σ (ou vocbulaire, ou lexique) est un ensemble fini de symboles. On note Σ^* l'ensemble des mots sur Σ.

Mot

Un mot (ou phrase) sur un vocabulaire Σ est une séquence finie d'éléments de Σ.

Alphabets

1. Σ={a, b} $\Rightarrow$ Σ^* ={a,b, ab,..., aaaa, aabb, ...}
2. {if, then, else, begin, end, :=, ;,(,), A, B, C, 1,2}

Mots

1. aaaa, aabb
2. if A then B := 1 else C := 2
   if if if A begin

Alphabets, Mots, et Langages

Alphabet

Un alphabet Σ (ou vocbulaire, ou lexique) est un ensemble fini de symboles. On note Σ^* l'ensemble des mots sur Σ.

Alphabets

1. Σ={a, b} $\Rightarrow$ Σ^* ={a,b, ab,..., aaaa, aabb, ...}
2. {if, then, else, begin, end, :=, ;,(,), A, B, C, 1,2}

Mot

Un mot (ou phrase) sur un vocabulaire Σ est une séquence finie d'éléments de Σ.

Mots

1. aaaa, aabb
2. if A then B := 1 else C := 2
   if if if A begin

Alphabets, Mots, et Langages (2)

Concaténation

$\bullet : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$
soit $x = x_1 x_2 \ldots x_k \in \Sigma^*$ et $y = y_1 y_2 \ldots y_l \in \Sigma^*$ alors $x.y = x_1 x_2 \ldots x_k y_1 y_2 \ldots y_l$ est la concaténation de x et y.

ε : mot vide, neutre pour la concaténation ($\varepsilon.w = w.\varepsilon = w$).
|u| longueur du mot u.
$|.| : \Sigma^* \rightarrow \mathbb{N}$ est le morphisme défini par $|a| = 1$ pour $a \in \Sigma$, $|\varepsilon|$=0.
$|u|_a$: nombre de a dans le mot u.
$W = w_1 w_2 \ldots w_k / w_i \in \Sigma \rightarrow |W|$=k

Définition : Langage

Soit $L \subseteq \Sigma^*$ on dira que L est un langage sur l'alphabet Σ.
On note {} ou $\emptyset$ le langage vide.

Langages sur $\Sigma = \{a, b\}$

L1= {a,abba, abbb, b}
L2={w / |w|=2k k $\in \mathbb{N}$}
={ε, aa, ab, ba, bb, aaaa, aaab, aaba, aabb, ...}

Alphabets, Mots, et Langages (2)

Concaténation

$\bullet : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$
soit $x = x_1x_2...x_k \in \Sigma^*$ et
$y = y_1y_2...y_l \in \Sigma^*$ alors
$x.y = x_1x_2...x_ky_1y_2...y_l$ est la
concaténation de x et y.

ε : mot vide, neutre pour la
concaténation ($\varepsilon.w = w.\varepsilon = w$).
|u| longueur du mot u.
$|.| : \Sigma^* \rightarrow \mathbb{N}$ est le morphisme défini
par |a| = 1 pour $a \in \Sigma$, $|\varepsilon|$=0.
$|u|_a$: nombre de a dans le mot u.
$W = w_1w_2...w_k / w_i \in \Sigma \rightarrow$ |W|=k

Définition : Langage

Soit $L \subseteq \Sigma^*$ on dira que L est
un langage sur l'alphabet Σ.
On note {} ou $\emptyset$ le langage vide.

Langages sur $\Sigma = \{a, b\}$

L1= {a,abba, abbb, b}
L2={w / |w|=2k k $\in \mathbb{N}$}
={ε, aa, ab, ba, bb, aaaa, aaab,
aaba, aabb, ...}

Alphabets, Mots, et Langages (2)

Concaténation

$\bullet : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$
soit $x = x_1 x_2 \ldots x_k \in \Sigma^*$ et
$y = y_1 y_2 \ldots y_l \in \Sigma^*$ alors
$x.y = x_1 x_2 \ldots x_k y_1 y_2 \ldots y_l$ est la concaténation de x et y.

ε : mot vide, neutre pour la concaténation $(\varepsilon.w = w.\varepsilon = w)$.
|u| longueur du mot u.
$|.| : \Sigma^* \rightarrow \mathbb{N}$ est le morphisme défini par |a| = 1 pour $a \in \Sigma$, $|\varepsilon|=0$.
$|u|_a$: nombre de a dans le mot u.
$W = w_1 w_2 \ldots w_k / w_i \in \Sigma \rightarrow |W|=k$

Définition : Langage

Soit $L \subseteq \Sigma^*$ on dira que L est un langage sur l'alphabet Σ.
On note {} ou $\emptyset$ le langage vide.

Langages sur $\Sigma = \{a, b\}$

L1= {a,abba, abbb, b}
L2={w / |w|=2k $k \in \mathbb{N}$}
={ε, aa, ab, ba, bb, aaaa, aaab, aaba, aabb, ...}

Alphabets, Mots, et Langages (2)

Concaténation

$\bullet : \Sigma^* \text{ x } \Sigma^* \mapsto \Sigma^*$
soit $x = x_1 x_2 ... x_k \in \Sigma^*$ et
$y = y_1 y_2 ... y_l \in \Sigma^*$ alors
$x.y = x_1 x_2 ... x_k y_1 y_2 ... y_l$ est la concaténation de x et y.

ε : mot vide, neutre pour la concaténation ($\varepsilon.w = w.\varepsilon = w$).
|u| longueur du mot u.
$|.| : \Sigma^* \rightarrow \mathbb{N}$ est le morphisme défini par |a| = 1 pour $a \in \Sigma$, $|\varepsilon|$=0.
$|u|_a$: nombre de a dans le mot u.
$W = w_1 w_2 ... w_k / w_i \in \Sigma \rightarrow$ |W|=k

Définition : Langage

Soit $L \subseteq \Sigma^*$ on dira que L est un langage sur l'alphabet Σ.
On note {} ou $\emptyset$ le langage vide.

Langages sur $\Sigma = \{a, b\}$

L1= {a,abba, abbb, b}
L2={w / |w|=2k k $\in \mathbb{N}$}
={ε, aa, ab, ba, bb, aaaa, aaab, aaba, aabb, ...}

Opérations sur les ensembles

- Union
- Intersection
- Complément
- Produit Cartésien
- Ensemble des parties
- Fermeture de Kleene

$A \cup B = \{w | w \in A \text{ ou } w \in B\}$

Ex. Soit $\Sigma = \{a, b\}$,
$L1 = \{\varepsilon, a, aa\}$ et $L2 = \{a, bb\}$.
$L1 \cup L2 = \{\varepsilon, a, aa, bb\}$

- R1 | R2 est une expression régulière correspondant à la **disjonction** de langues.
- R1* est une expression régulière correspondant à la fermeture de **Kleene** du langue.
- (R) est une expression régulière correspondant à R.

Opérations sur les ensembles

- Union
- Intersection
- Complément
- Produit Cartésien
- Ensemble des parties
- Fermeture de Kleene

$A \cap B = \{w | w \in A \text{ et } w \in B\}$

Ex. Soit $\Sigma = \{a, b\}$
$L1 = \{\varepsilon, a, aa\}$ et $L2 = \{a, bb\}$.
$L1 \cap L2 = \{a\}$

- R1 | R2 est une expression régulière correspondant à la **disjonction** de langues.
- R1* est une expression régulière correspondant à la fermeture de **Kleene** du langue.
- (R) est une expression régulière correspondant à R.

Opérations sur les ensembles

- Union
- Intersection
- Complément
- Produit Cartésien
- Ensemble des parties
- Fermeture de Kleene

$\overline{A} = \{w | w \notin A\}$

plus préciséiment, soit U l'ensemble de réference, alors $\overline{A} = \{w | w \in U \text{ et } w \notin A\}$

Ex. Soit $\Sigma = \{a, b\}$ et $L_1 = \{\varepsilon, a, aa\}$
$\overline{L_1} = \{b, ab, ba, bb, aaa, aab, aba, baa, bba, ...\}$

- R1 | R2 est une expression régulière correspondant à la **disjonction** de langues.
- R1* est une expression régulière correspondant à la fermeture de **Kleene** du langue.
- (R) est une expression régulière correspondant à R.

Opérations sur les ensembles

- Union
- Intersection
- Complément
- Produit Cartésien
- Ensemble des parties
- Fermeture de Kleene

$A \times B = \{x \in A \text{ et } y \in B\}$

Note : A x ∅ =∅ ; |A x B| =|A||B|

Ex. Soit $\Sigma = \{a, b\}$, $L_1 = \{\varepsilon, a, aa\}$ et $L_2 = \{a, bb\}$.
$L_1 \times L_2 = \{(\varepsilon, a), (a, a), (a, bb), (aa, a), (aa, bb)\}$

- R1 | R2 est une expression régulière correspondant à la **disjonction** de langues.
- R1* est une expression régulière correspondant à la fermeture de **Kleene** du langue.
- (R) est une expression régulière correspondant à R.

Opérations sur les ensembles

- Union
- Intersection
- Complément
- Produit Cartésien
- Ensemble des parties
- Fermeture de Kleene

$\mathcal{P}(A) = \{B | B \subseteq A\}$

Note : $|\mathcal{P}(A)| = 2^{|A|}$

Ex. $L_1 = \{\varepsilon, a, aa\}$
$\mathcal{P}(L_1) =$
$\{\emptyset, \{\varepsilon\}, \{a\}, \{aa\}, \{\varepsilon, a\}, \{\varepsilon, aa\}, \{a, aa\}, \{\varepsilon, a, aa\}\}$

- R1 | R2 est une expression régulière correspondant à la **disjonction** de langues.
- R1* est une expression régulière correspondant à la fermeture de **Kleene** du langue.
- (R) est une expression régulière correspondant à R.

Opérations sur les ensembles

- Union
- Intersection
- Complément
- Produit Cartésien
- Ensemble des parties
- Fermeture de Kleene

Soit L un langage sur l'alphabet Σ alors $L^* = \{w = x_1.x_2......x_k | k \geq 0 \ et \ x_i \in L\}$ est la fermeture de Kleene de L, dénotée par l'ER (a)* | $a \in L$. Ex. $L_2 = \{a, bb\}$.

$L_2^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, ...\}$

Note : pour tout langage $L \neq \emptyset$ on a $|L^*| = \infty$

- R1 | R2 est une expression régulière correspondant à la **disjonction** de langues.
- R1* est une expression régulière correspondant à la fermeture de **Kleene** du langue.
- (R) est une expression régulière correspondant à R.

Expressions régulières en Action

Example :

Nombres paire (+|-| ϵ) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

Simplification de notation : Définitions régulières

Signe = + | -
OptSigne = Signe | ϵ
chiffre = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
chiffrePaire= 0 | 2 | 4 | 6 | 8
nombrePaire= OptSigne chiffre* chiffrePaire

Expressions régulières en Action

Example :

Nombres paire $(+|\text{-}|\ \epsilon)$ $(0|1|2|3|4|5|6|7|8|9)^*$ $(0|2|4|6|8)$

Simplification de notation : Disjonction multi-manières

Signe = + | -
OptSigne = Signe | ϵ
chiffre = [0123456789]
chiffrePaire=[02468]
nombrePaire= OptSigne chiffre* chiffrePaire

Expressions régulières en Action

Example :

Nombres paire (+|-| ϵ) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

Simplification de notation : Ranges

Signe = + | -
OptSigne = Signe | ϵ
chiffre = [0-9]
chiffrePaire= [02468]
nombrePaire= OptSigne chiffre* chiffrePaire

Expressions régulières en Action

Example :

Nombres paire $(+|\text{-}|\ \epsilon)$ (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

Simplification de notation : Ranges

Signe = + | -
OptSigne = Signe | ϵ
chiffre = [0-9]
chiffrePaire= [02468]
nombrePaire= OptSigne chiffre* chiffrePaire

Expressions régulières en Action

Example :

Nombres paire (+|-| ϵ) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

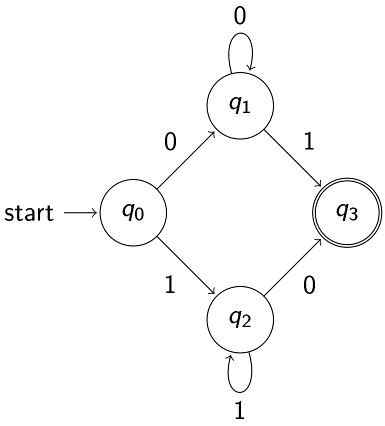
Simplification de notation : zero-ou-un

Signe = + | -
OptSigne = Signe ?
chiffre = [0-9]
chiffrePaire= [02468]
nombrePaire= OptSigne chiffre* chiffrePaire

La mise en œuvre des expressions régulières

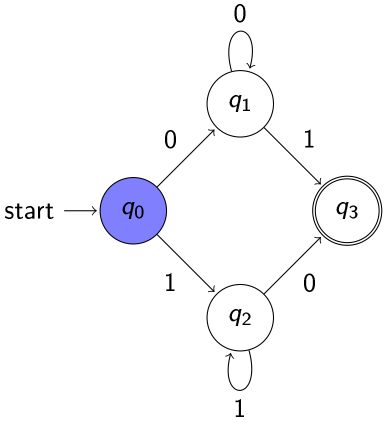
- Les expressions régulières peuvent être mis en œuvre à l'aide **automates finis**.
- Il existe deux types d'automates finis :
  - AFD (**automates finis déterministes**), que nous le verrons dans un peu, et
  - AFN (**automates finis non déterministe**), que nous allons voir plus loin ;
- Les automates sont mieux expliquées par l'exemple ...

Exemple d'utilisation d'AF



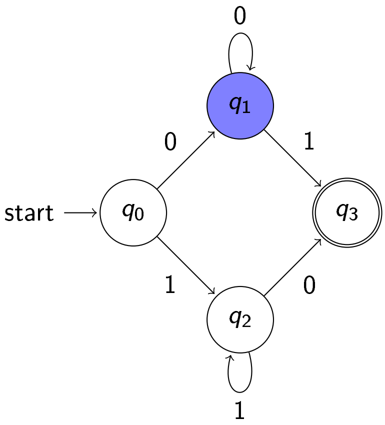
0	0	0	1

Exemple d'utilisation d'AF



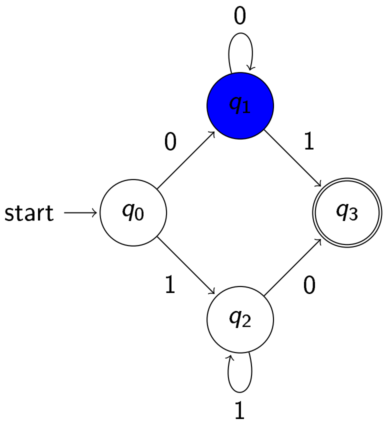
0	0	0	1
$\Uparrow$			

Exemple d'utilisation d'AF



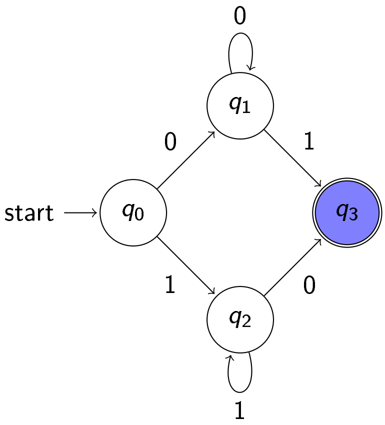
0	0	0	1
	$\Uparrow$		

Exemple d'utilisation d'AF



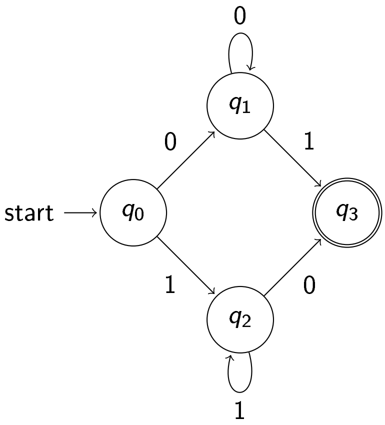
0	0	0	1
		⇑	

Exemple d'utilisation d'AF



0	0	0	1
			$\Uparrow$

Exemple d'utilisation d'AF



0	0	0	1

OK

Automates Finis Déterministes

Automate Fini (AFD)

Un automate fini (AFD) est un quintuplet $(Q, \Sigma, \delta, q_0, F)$
- Q, un ensemble fini d'états ;
- Σ un alphabet (fini) ;
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \rightarrow Q$ une fonction de transition ;
- $q_0 \in Q$ un état initial ;
- $F \subseteq Q$, des états finaux (accepteurs).

Exemple

$\Sigma = \{a, b\}$
$L(A) = \{w : |w|a \text{ est pair}\}$.
$|L(A)| = \infty$
$A = (Q, \Sigma, \delta, q_0, F)$ où
$Q = \{q_0, q_1\}$
$F = \{q_0\}$
et δ est donnée formellement par :

	a	b
$\rightleftharpoons$ q0	q1	q0
q1	q0	q1

L'automate prend en entrée un mot et l'accepte ou le rejette. On dit aussi qu'il le reconnaît ou ne le reconnaît pas.

q0 : état d'acceptation

Automates Finis Déterministes

Automate Fini (AFD)

Un automate fini (AFD) est un quintuplet $(Q, \Sigma, \delta, q_0, F)$
- Q, un ensemble fini d'états ;
- Σ un alphabet (fini) ;
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \rightarrow Q$ une fonction de transition ;
- $q_0 \in Q$ un état initial ;
- $F \subseteq Q$, des états finaux (accepteurs).

Exemple

$\Sigma = \{a, b\}$
$L(A) = \{w : |w|a \text{ est pair}\}$.
$|L(A)| = \infty$
$A = (Q, \Sigma, \delta, q_0, F)$ où
$Q = \{q_0, q_1\}$
$F = \{q_0\}$
et δ est donnée formellement par :

	a	b
$\rightleftharpoons$ q0	q1	q0
q1	q0	q1

L'automate prend en entrée un mot et l'accepte ou le rejette. On dit aussi qu'il le reconnaît ou ne le reconnaît pas.

q0 : état d'acceptation

Automates Finis Déterministes

Automate Fini (AFD)

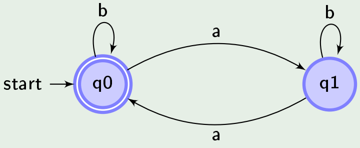
Un automate fini (AFD) est un quintuplet $(Q, \Sigma, \delta, q_0, F)$
- Q, un ensemble fini d'états ;
- Σ un alphabet (fini) ;
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \rightarrow Q$ une fonction de transition ;
- $q_0 \in Q$ un état initial ;
- $F \subseteq Q$, des états finaux (accepteurs).

Exemple

$\Sigma = \{a, b\}$
$L(A) = \{w : |w|a \text{ est pair}\}$.
$|L(A)| = \infty$
$A = (Q, \Sigma, \delta, q_0, F)$ où
$Q = \{q_0, q_1\}$
$F = \{q_0\}$
et δ est donnée formellement par :

	a	b
$\rightleftharpoons$ q0	q1	q0
q1	q0	q1

L'automate prend en entrée un mot et l'accepte ou le rejette. On dit aussi qu'il le reconnaît ou ne le reconnaît pas.



q0 : état d'acceptation

Langages réguliers

Soit $A = (Q, \Sigma, \delta, q_0, F)$ un AFD

Définition (état accessible)

Un état $q \in Q$ est accessible s'il existe $w \in \Sigma^*$ tel que $(w, q0) \overset{*}{\vdash} (\varepsilon, q)$.

Définition (Mot accepté)

Soit $w \in \Sigma^*$. On dira que A accepte w ssi $\exists q \in F$ tel que $(w, q0) \overset{*}{\vdash} (\varepsilon, q)$

Définition (langage accepté)

Soit A un AF. $L(A) = \{w : \ A \ accepte \ w\}$ est le langage accepté par A.

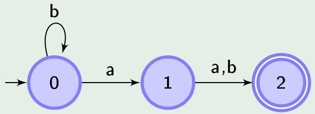
Définition (langage régulier)

Un langage Y est régulier (R) ssi il existe un AF tel que Y = L(A).

AF Non déterminisme

- Un automate A est déterministe si pour toute configuration de A, il existe au plus un mouvement possible.

Déterministe



- Un automate estnon déterministe s'il existe des configurations, pour lesquelles plus d'un mouvement est possible.
- Deux origines :
  - transitions-ε
  - $+$ d'1 Movement possible

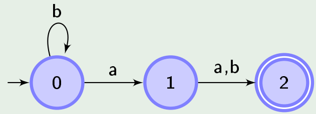
Non déterministe

AF Non déterminisme

- Un automate A est déterministe si pour toute configuration de A, il existe au plus un mouvement possible.

- Un automate estnon déterministe s'il existe des configurations, pour lesquelles plus d'un mouvement est possible.
- Deux origines :
  - transitions-ε
  - + d'1 Movement possible

Déterministe

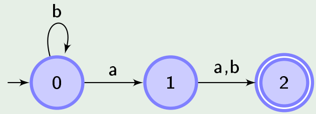


Non déterministe

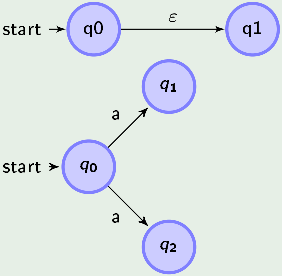
AF Non déterminisme

- Un automate A est déterministe si pour toute configuration de A, il existe au plus un mouvement possible.
- Un automate estnon déterministe s'il existe des configurations, pour lesquelles plus d'un mouvement est possible.
- Deux origines :
  - transitions-ε
  - + d'1 Movement possible

Déterministe

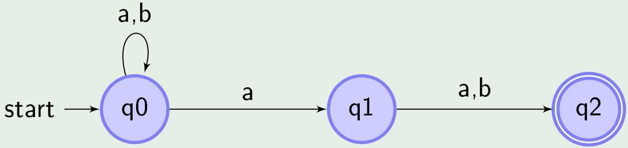


Non déterministe



Exemple d'un automate non déterministe

Automate non déterministe (AFN)



(Attention !) δ est donné formellement par :

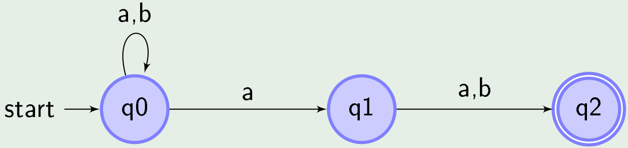
	a	b
$\rightarrow$ q0	{q0,q1}	{q0}
q1	{q2}	{q2}
$\leftarrow$ q2	$\emptyset$	$\emptyset$

Remarque

Un automate fini déterministe (AFD) est un cas particulier d'un AFN :

Exemple d'un automate non déterministe

Automate non déterministe (AFN)



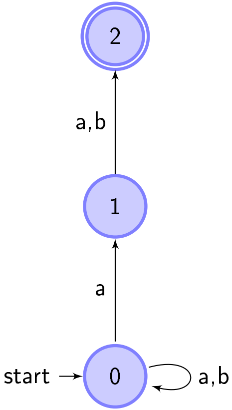
(Attention !) δ est donné formellement par :

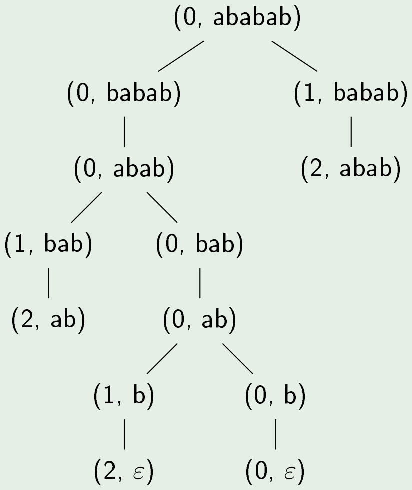
	a	b
$\rightarrow$ q0	{q0,q1}	{q0}
q1	{q2}	{q2}
$\leftarrow$ q2	$\emptyset$	$\emptyset$

Remarque

Un automate fini déterministe (AFD) est un cas particulier d'un AFN :

Reconnaissance d'un mot par un AFN





Lignes directrices

Déterminisation

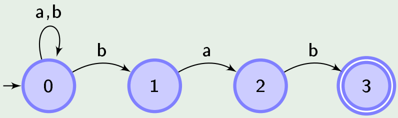
Théorème de Rabin-Scott

Considérons un automate fini non-déterministe $A_n = (Q_n, \Sigma, \delta_n, q_0, F_n)$ et construisons un automate fini déterministe $A_d = (Q_d, \Sigma, \delta_d, \{q_0\}, F_d)$ qui reconnaît exactement le même langage.

- Les alphabets de A_n et de A_d sont identiques.
- Les états de départ sont respectivement q_0 et le singleton $\{q_0\}$.
- Q_d est constitué de tous les sous-ensembles de Q_n.
- F_d est l'ensemble des sous-ensembles de Q_n qui contiennent au moins un élément de F_n.
- Étant donné un sous ensemble S de Q_n et un symbole $a \in \Sigma$, on définit la fonction de transition $\delta_d(S, a)$ de la manière suivante : $\delta_d(S, a) = \bigcup_{q \in S} \delta_n(q, a)$

Déterminisation (Exemple)

Soit A l'automate fini non-déterministe reconnaissant les mots de l'alphabet a,b qui terminent par bab.

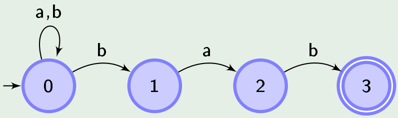


Pour déterminiser A en construisant un nouvel état à partir de chaque sous ensemble d'état possible.

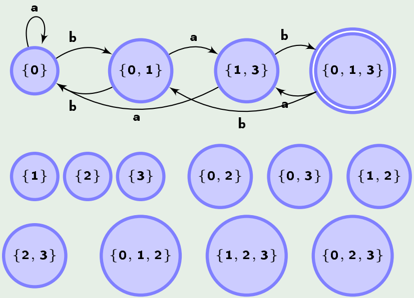
Remarquons que les états {1}, {2}, {3}, {0,2}, {0,3}, {1,2}, {2,3}, {0,1,2}, {1,2,3}, {0,2,3} sont inatteignables et peuvent être "retirés" de l'automate.

DÉTERMINISATION (EXEMPLE)

Soint A l'automate fini non-déterministe reconnaissant les mots de l'alphabet a,b qui terminent par bab.



Pour déterminiser A en construisant un nouvel état à partir de chaque sous ensemble d'état possible.



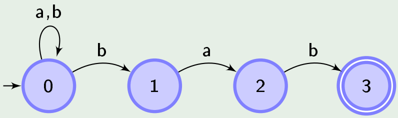
Remarquons que les états {1}, {2}, {3}, {0,2}, {0,3}, {1,2}, {2,3}, {0,1,2}, {1,2,3}, {0,2,3} sont inatteignables et peuvent être "retirés" de l'automate.

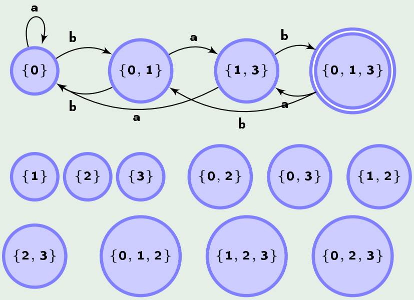
Déterminisation (Exemple)

Soint A l'automate fini non-déterministe reconnaissant les mots de l'alphabet a,b qui terminent par bab.

Pour déterminiser A en construisant un nouvel état à partir de chaque sous ensemble d'état possible.

Remarquons que les états {1}, {2}, {3}, {0,2}, {0,3}, {1,2}, {2,3}, {0,1,2}, {1,2,3}, {0,2,3} sont inatteignables et peuvent être "retirés" de l'automate.





DÉTERMINISATION (EN PRATIQUE)

Lors de la déterminisation d'un AFN, on ne crée pas immédiatement tous les états de l'AFD.
Les états "utiles" sont crées quand on en a besoin en suivant la méthode de construction ci-dessous :

- Q_d est initialisé à $\emptyset$ et soit E un ensemble d'états initialisé à $E = \{\{q0\}\}$
- Tant que E est non vide,
  - choisir un élément S de E (S est donc un sous ensemble de Q_n),
  - ajouter S à Q_d ,
  - pour tout symbole $a \in \Sigma$,
    - calculer l'état $S' = \bigcup_{q \in S} \delta_n(q, a)$
    - si S' n'est pas déjà dans Q_d , l'ajouter à E
    - ajouter un arc sur l'automate entre S et S´ et la valuer par a.

Déterminisation (Mise en œuvre)

- Dans la pratique, lorsque l'on veut construire un automate déterministe AFD à partir d'un automate non déterministe sans transitions-ε AFN, on ne commence pas par créer tous les états de AFD, car ils peuvent être nombreux : $|p(Q)| = 2^{|Q|}$!
- On construit pluôt les états de AFD au fur et mesure de leur création en partant de l'état initial.

Exemple

	a	b
→0	$\{0,1\}$	0
1	2	2
←2	$\emptyset$	$\emptyset$

On part de l'état initial 0 dont on calcule les transitions :

	a	b
→0	01	0

un nouvel état 01 a été crée, dont on calcule les transition.

Ce calcul mène à la création des deux nouveaux états 01 et 012 qui ne donneront pas naissance à de nouveaux états, ce qui nous donne l'automate déterministe suivant :

	a	b
→0	01	0
01	012	02
←02	01	0
←012	012	02

Déterminisation (Mise en œuvre)

- Dans la pratique, lorsque l'on veut construire un automate déterministe AFD à partir d'un automate non déterministe sans transitions-ε AFN, on ne commence pas par créer tous les états de AFD, car ils peuvent être nombreux : $|p(Q)| = 2^{|Q|}$!
- On construit pluôt les états de AFD au fur et mesure de leur création en partant de l'état initial.

Exemple

	a	b
$\rightarrow$0	$\{0,1\}$	0
1	2	2
$\leftarrow$2	$\emptyset$	$\emptyset$

On part de l'état initial 0 dont on calcule les transitions :

	a	b
$\rightarrow$0	01	0

un nouvel état 01 a été crée, dont on calcule les transition.

Ce calcul mène à la création des deux nouveaux états 01 et 012 qui ne donneront pas naissance à de nouveaux états, ce qui nous donne l'automate déterministe suivant :

	a	b
$\rightarrow$0	01	0
01	012	02
$\leftarrow$02	01	0
$\leftarrow$012	012	02

Déterminisation (Mise en œuvre)

- Dans la pratique, lorsque l'on veut construire un automate déterministe AFD à partir d'un automate non déterministe sans transitions-ε AFN, on ne commence pas par créer tous les états de AFD, car ils peuvent être nombreux : $|p(Q)| = 2^{|Q|}$!
- On construit pluôt les états de AFD au fur et mesure de leur création en partant de l'état initial.

Exemple

	a	b
$\rightarrow$0	$\{0,1\}$	0
1	2	2
$\leftarrow$2	$\emptyset$	$\emptyset$

On part de l'état initial 0 dont on calcule les transitions :

	a	b
$\rightarrow$0	01	0

un nouvel état 01 a été crée, dont on calcule les transition.

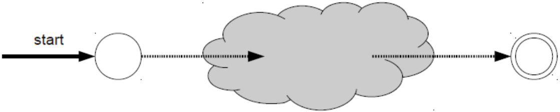
Ce calcul mène à la création des deux nouveaux états 01 et 012 qui ne donneront pas naissance à de nouveaux états, ce qui nous donne l'automate déterministe suivant :

	a	b
$\rightarrow$0	01	0
01	012	02
$\leftarrow$02	01	0
$\leftarrow$012	012	02

Lignes directrices

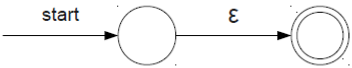
Conversion d'une ER en AFN

- Il existe une procédure (magnifique) qui permet de convertir une expression régulière en AFN.
- Associez chaque expression régulière à un AFN avec les propriétés suivantes :
  - Il y a exactement un état acceptant.
  - Il n'y a pas de transitions de l'état acceptant.
  - Il n'y a pas de transitions dans l'état de départ.

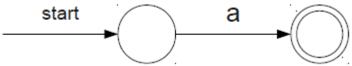


Les cas de base

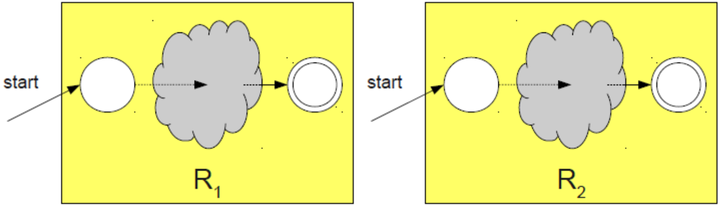
pour ϵ



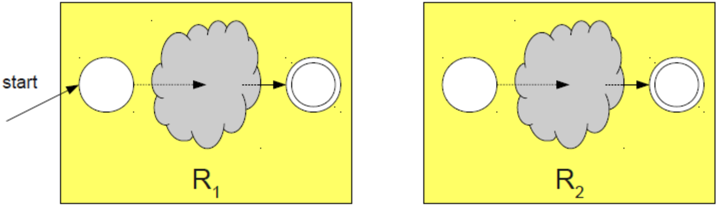
pour a



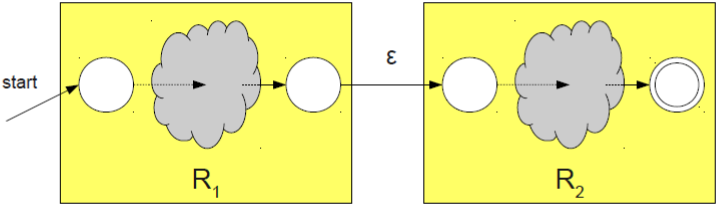
Construction pour R1R2



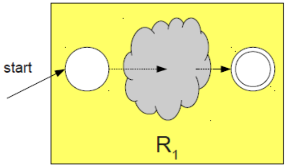
Construction pour R1R2

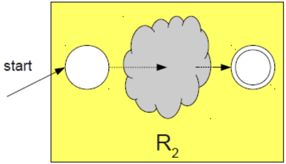


Construction pour R1R2

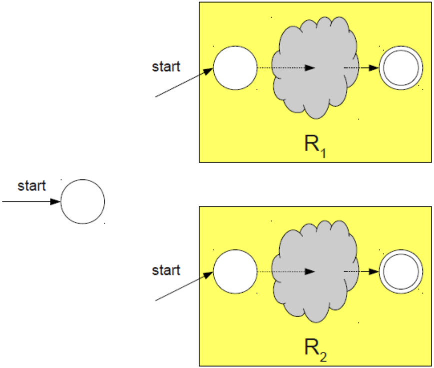


Construction pour R1|R2

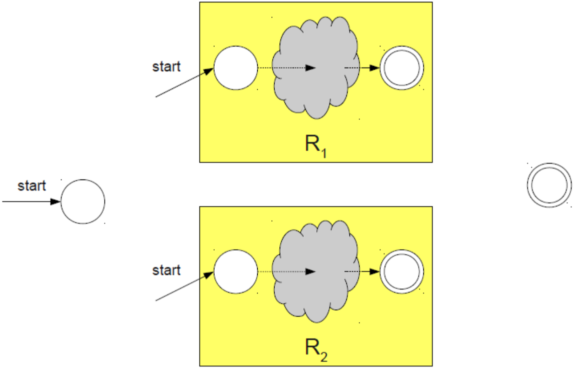




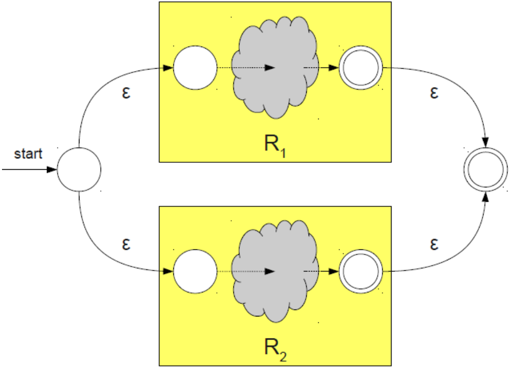
Construction pour R1|R2



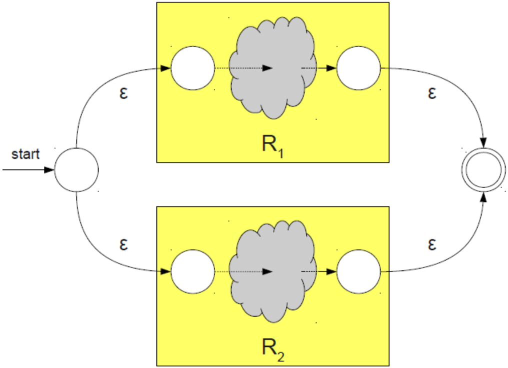
Construction pour R1|R2



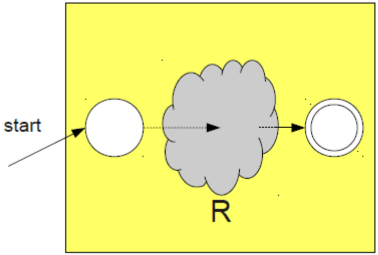
Construction pour R1|R2



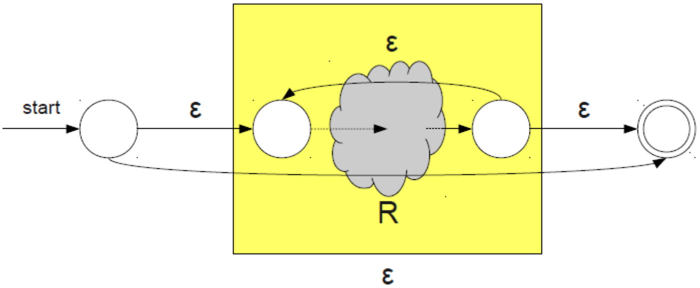
Construction pour R1|R2



Construction pour R^*



Construction pour R*



Ambiguïté Lexicale

Ambiguïté Lexicale

T_For for
T_Identifier [A-Za-z_][A-Za-z0-9_]*

Mot : fort

f	o	r	t		
f	o	r		t	
f	o	r		t	
f	o		r	t	
f	o		r		t

f		o	r	t		
f	o		r		t	
f		o		r	t	
f		o	r		t	
f		o		r		t

Ambiguïté Lexicale (longest match)

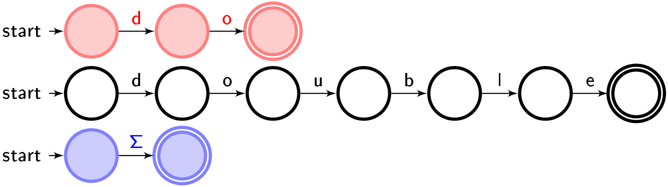
Solution : Maximal Munch

T_For	for
T_Identifier	[A-Za-z_][A-Za-z0-9_]*

f o r t

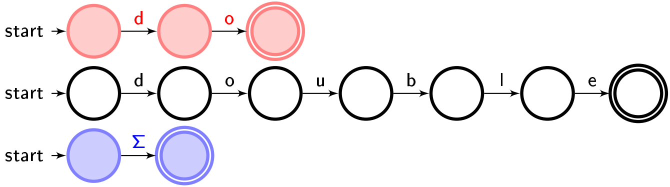
Implantation du Maximal Munch (1)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Implantation du Maximal Munch (1)

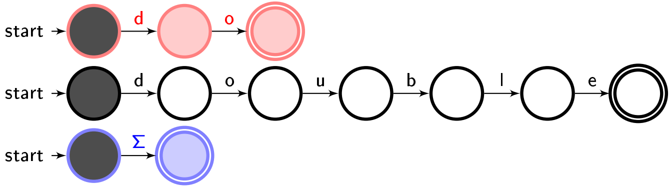
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



D	O	U	B	D	O	U	B	L	E

Implantation du Maximal Munch (2)

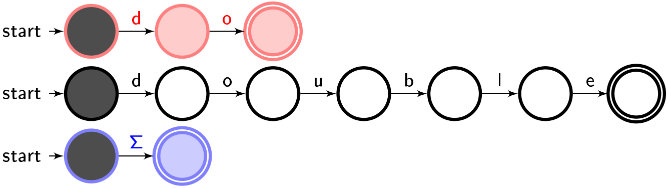
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



D	O	U	B	D	O	U	B	L	E

Implantation du Maximal Munch (2)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]

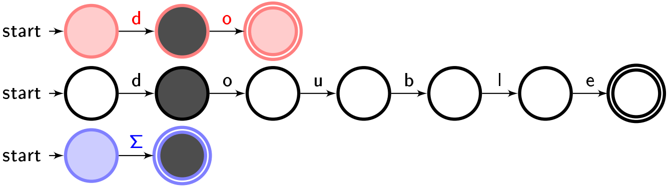


D	O	U	B	D	O	U	B	L	E

⇑

Implantation du Maximal Munch (3)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]

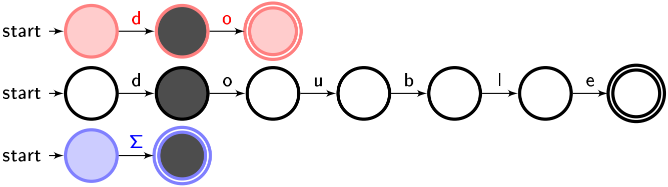


D	O	U	B	D	O	U	B	L	E

⇑

Implantation du Maximal Munch (3)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



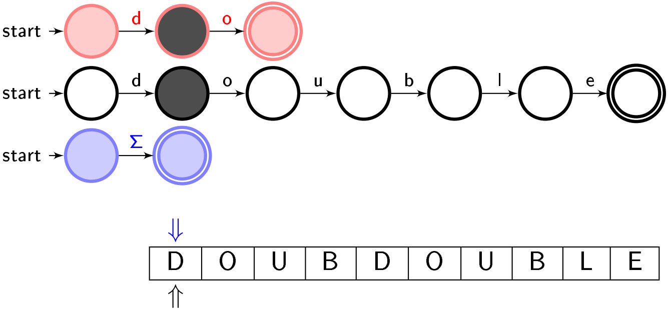
$\Downarrow$

D	O	U	B	D	O	U	B	L	E

$\Uparrow$

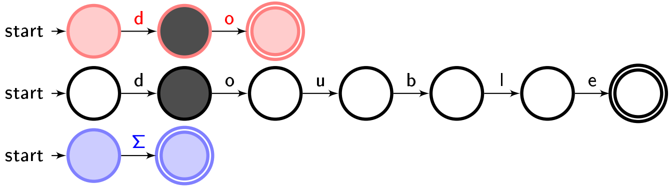
Implantation du Maximal Munch (4)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Implantation du Maximal Munch (4)

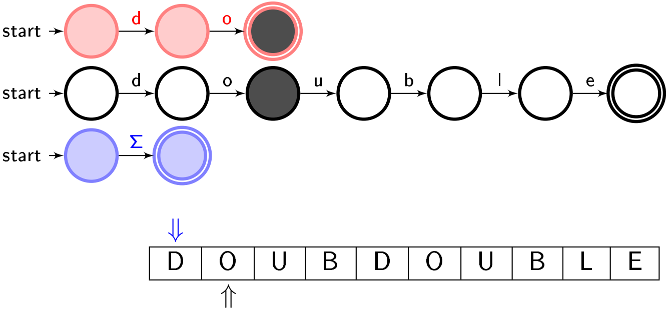
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



D	O	U	B	D	O	U	B	L	E

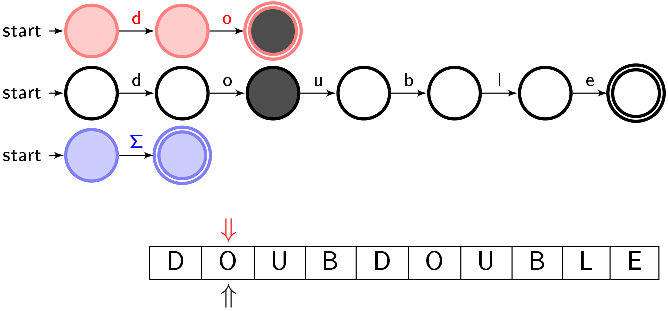
Implantation du Maximal Munch (5)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



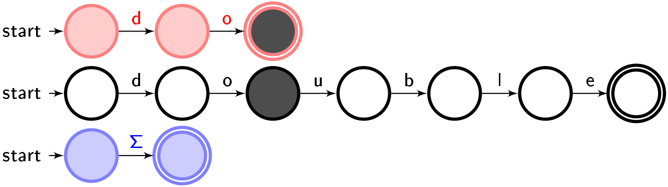
Implantation du Maximal Munch (5)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Implantation du Maximal Munch (5)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



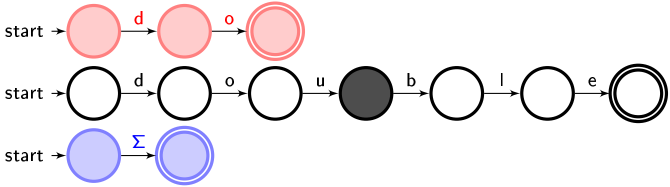
$\Downarrow$

D	O	U	B	D	O	U	B	L	E

$\Uparrow$

Implantation du Maximal Munch (6)

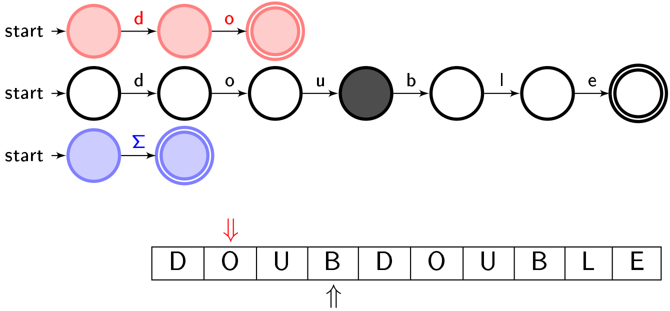
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



D	O	U	B	D	O	U	B	L	E

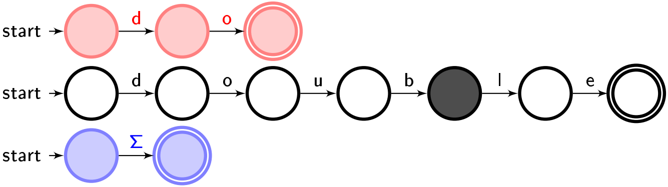
Implantation du Maximal Munch (6)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Implantation du Maximal Munch (7)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



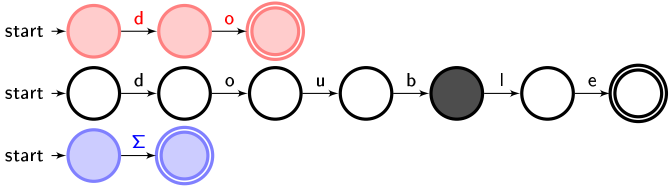
⇓

D	O	U	B	D	O	U	B	L	E

⇑

Implantation du Maximal Munch (7)

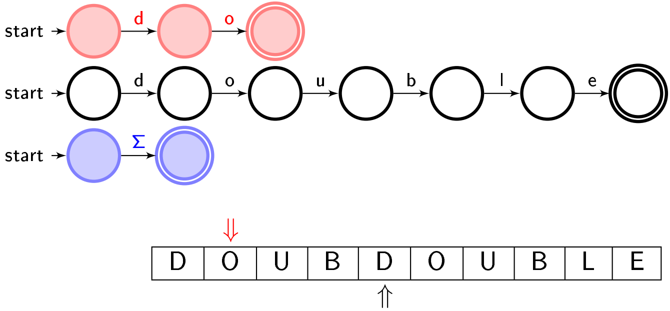
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



D	O	U	B	D	O	U	B	L	E

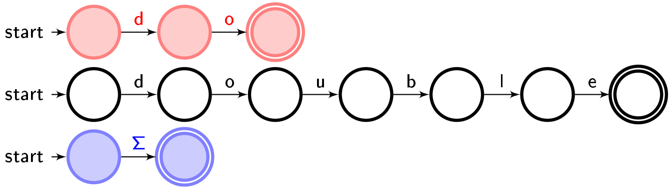
Implantation du Maximal Munch (8)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Implantation du Maximal Munch (9)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



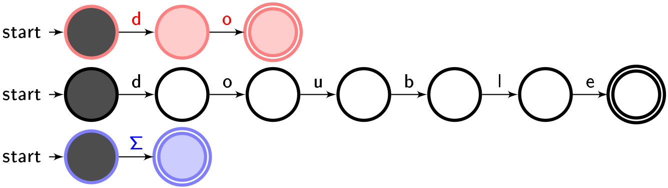
D	O

U	B	D	O	U	B	L	E

⇑

Implantation du Maximal Munch (10)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



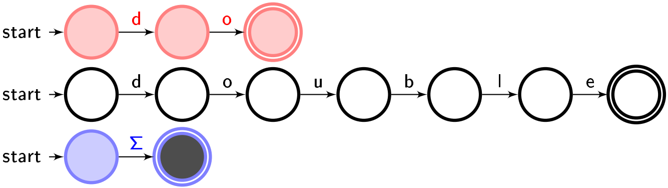
D	O

U	B	D	O	U	B	L	E

⇑

Implantation du Maximal Munch (11)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



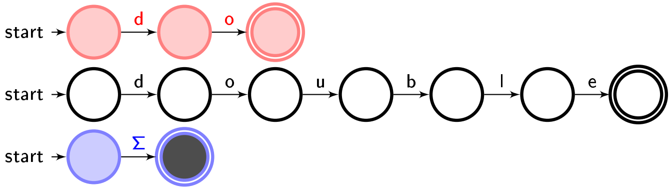
D	O

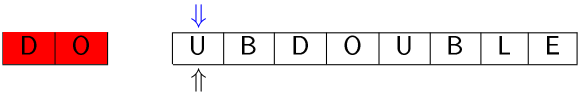
U	B	D	O	U	B	L	E

⇑

Implantation du Maximal Munch (11)

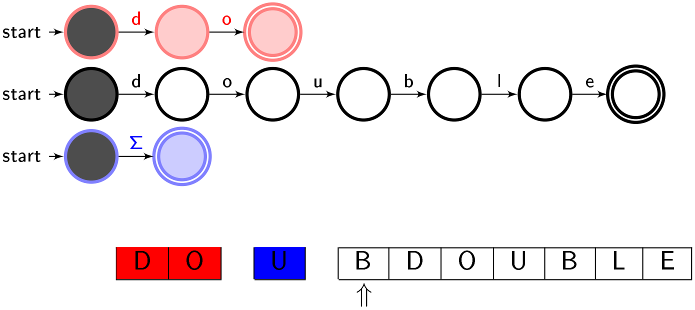
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]





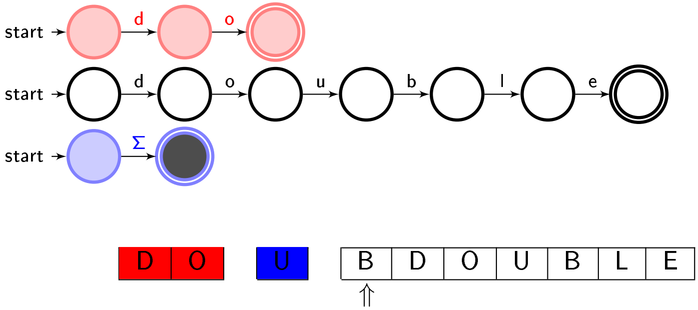
Implantation du Maximal Munch (12)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



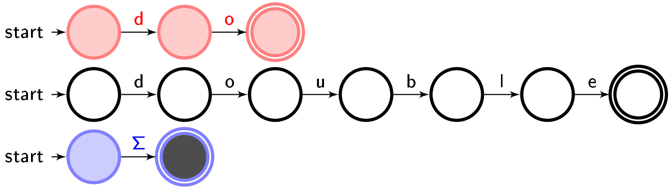
Implantation du Maximal Munch (12)

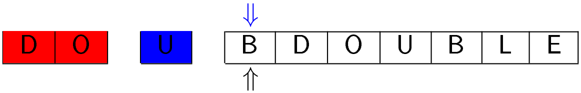
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Implantation du Maximal Munch (12)

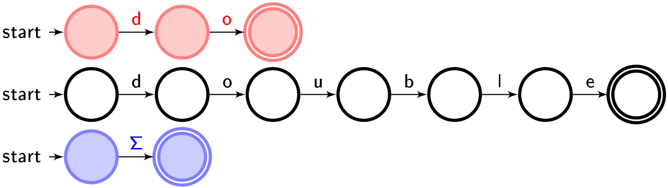
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]

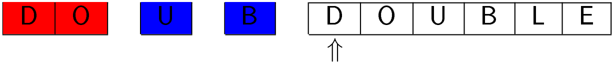




Implantation du Maximal Munch (13)

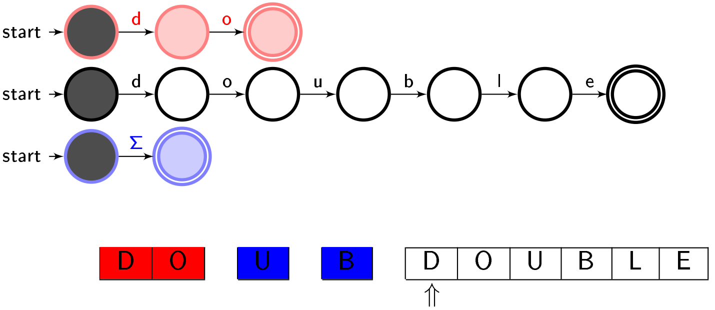
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]





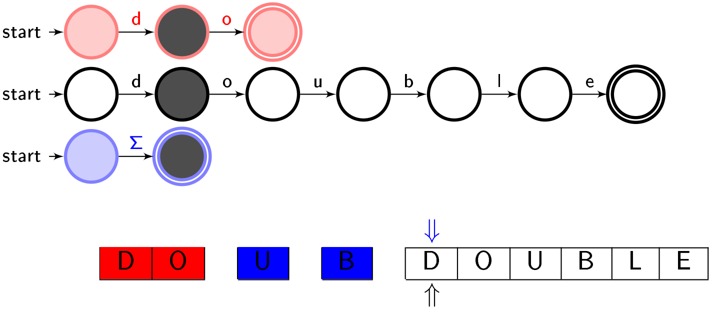
Implantation du Maximal Munch (14)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



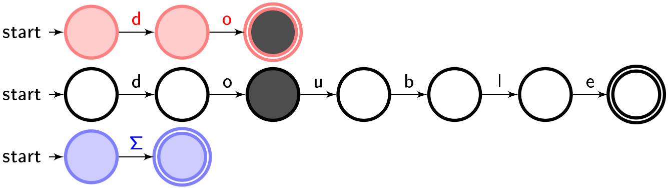
Implantation du Maximal Munch (15)

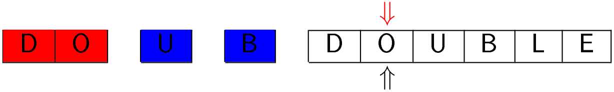
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Implantation du Maximal Munch (16)

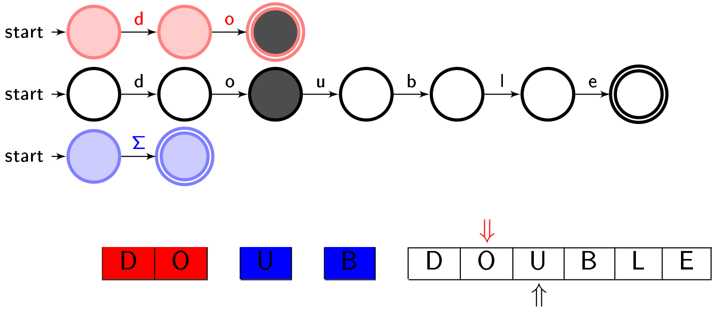
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]





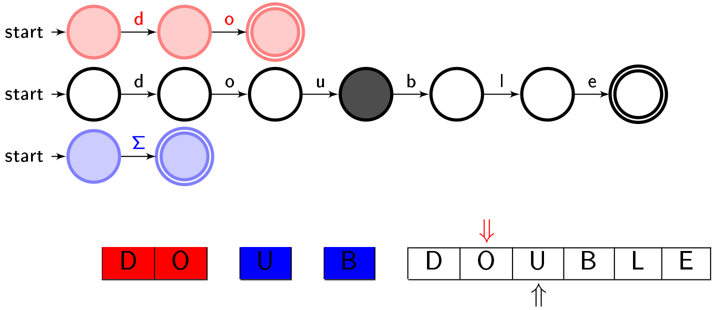
Implantation du Maximal Munch (16)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



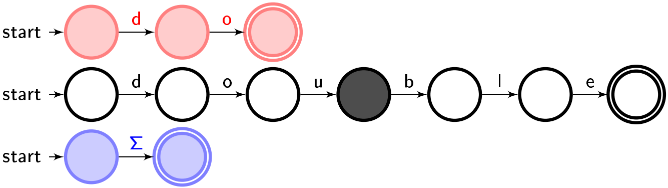
Implantation du Maximal Munch (17)

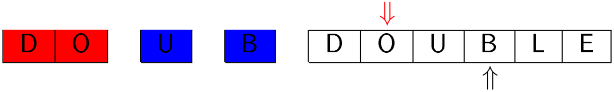
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Implantation du Maximal Munch (17)

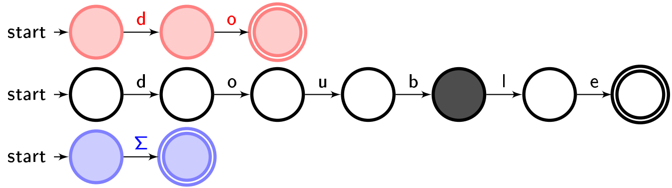
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]

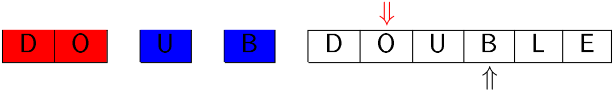




Implantation du Maximal Munch (18)

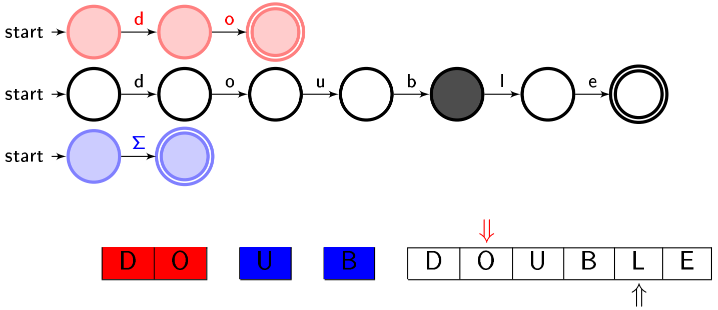
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]





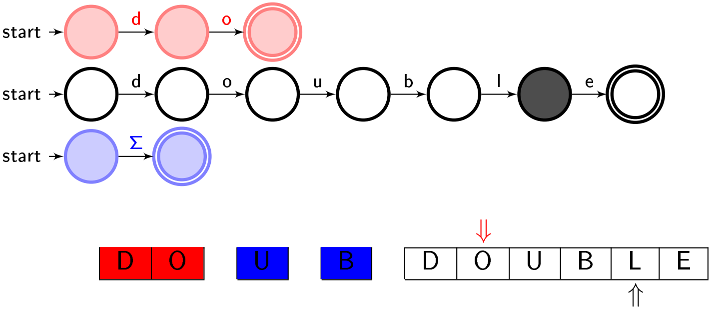
Implantation du Maximal Munch (18)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



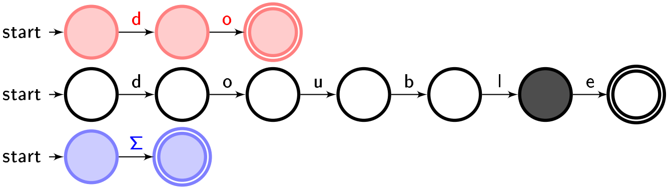
Implantation du Maximal Munch (19)

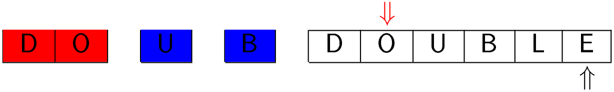
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Implantation du Maximal Munch (19)

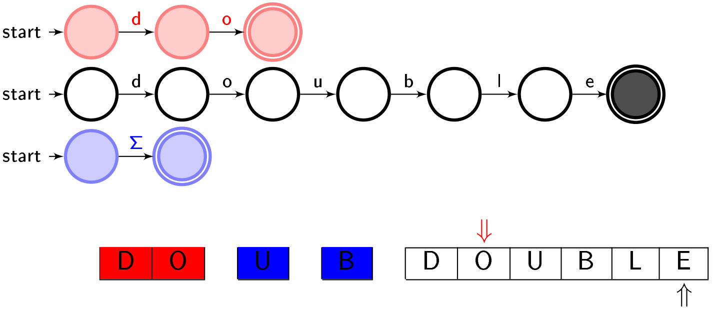
T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]





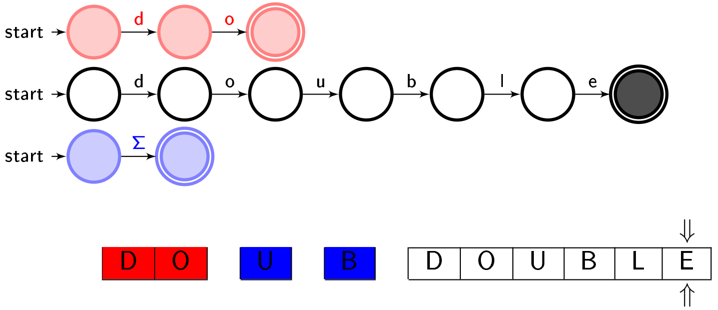
Implantation du Maximal Munch(20)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



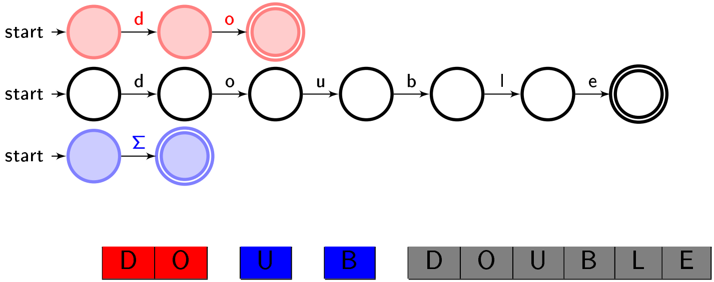
Implantation du Maximal Munch(20)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]

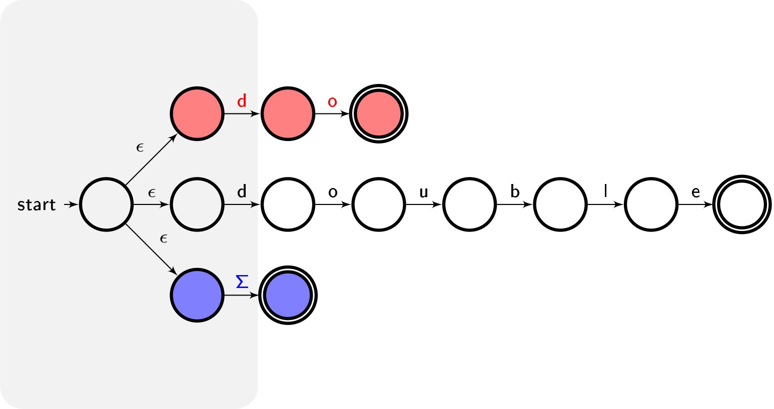


Implantation du Maximal Munch(20)

T_Do	do
T_Double	double
T_Mystere (inconnu)	[A-Za-z]



Petite Simplification



Un exemple de classe de lexèmes de JAVA est :

IDENT	foo, main,
NUMBER	0, 123, 1000
FLOAT	0.5, 1.0e+3

Analyseur lexical en dur (introduction)

Pour le Programme

```
class Foo {
  void bar {
    println ("hello world\n") ;
  }
}
```

La sortie du Scanner

CLASS IDENT(Foo) LBRACE VOID IDENT(bar) LPAREN RPAREN LBRACE IDENT(println) LPAREN STRING ("hello world\n") RPAREN SEMICOLON RBRACE RBRACE EOF

Sortie du Scanner ⇒ entrée du Parser.

L'interface entre l'analyseur lexical et l'analyseur syntaxique doit être une fonction (par exemple *int nexSym()*), qui renvoie à chaque appel l'unité lexicale (symbole) suivante trouvée dans le texte source.

nextSym()

```
Token sym ;
void nextSym ()
  "ignore les espaces blancs et assigne le prochain lexème à sym"
    ...
}
```

Les deux analyseurs partagent la définitions de symboles.

```
interface Symbols {
  static final int ERROR = 0, EOF = ERROR + 1, IDENT = EOF + 1,
    LITERAL = IDENT + 1, LPAREN = LITERAL + 1, RPAREN = LPAREN + 1,
```

Analyseur lexical en dur (introduction)

Pour le Programme

```
class Foo {
    void bar {
        println ("hello world\n") ;
    }
}
```

La sortie du Scanner

CLASS IDENT(Foo) LBRACE VOID IDENT(bar) LPAREN RPAREN LBRACE IDENT(println) LPAREN STRING ("hello world\n") RPAREN SEMICOLON RBRACE RBRACE EOF

Sortie du Scanner ⇒ entrée du Parser.

L'interface entre l'analyseur lexical et l'analyseur syntaxique doit être une fonction (par exemple *int nexSym()*), qui renvoie à chaque appel l'unité lexicale (symbole) suivante trouvée dans le texte source.

nextSym()

```
Token sym ;
void nextSym ()
    "ignore les espaces blancs et assigne le prochain lexème à sym"
    ...
}
```

Les deux analyseurs partagent la définitions de symboles.

```
interface Symbols {
    static final int ERROR = 0, EOF = ERROR + 1, IDENT = EOF + 1,
      LITERAL = IDENT + 1, LPAREN = LITERAL + 1, RPAREN = LPAREN + 1,
```

Analyseur lexical en dur (introduction)

Pour le Programme

```
class Foo {
   void bar {
      println ("hello world\n") ;
   }
}
```

La sortie du Scanner

CLASS IDENT(Foo) LBRACE VOID IDENT(bar) LPAREN RPAREN LBRACE IDENT(println) LPAREN STRING ("hello world\n") RPAREN SEMICOLON RBRACE RBRACE EOF

Sortie du Scanner ⇒ entrée du Parser.

L'interface entre l'analyseur lexical et l'analyseur syntaxique doit être une fonction (par exemple ***int nexSym()***), qui renvoie à chaque appel l'unité lexicale (symbole) suivante trouvée dans le texte source.

nextSym()

```
   Token sym ;
   void nextSym ()
      "ignore les espaces blancs et assigne le
prochain lexème à sym"
         ...

   }
```

Les deux analyseurs partagent la définitions de symboles.

```
interface Symbols {
    static final int ERROR = 0, EOF = ERROR + 1, IDENT = EOF + 1,
      LITERAL = IDENT + 1, LPAREN = LITERAL + 1, RPAREN = LPAREN + 1,
```

Analyseur lexical en dur (introduction)

Pour le Programme

```
class Foo {
  void bar {
    println ("hello world\n") ;
  }
}
```

La sortie du Scanner

CLASS IDENT(Foo) LBRACE VOID IDENT(bar) LPAREN RPAREN LBRACE IDENT(println) LPAREN STRING ("hello world\n") RPAREN SEMICOLON RBRACE RBRACE EOF

Sortie du Scanner ⇒ entrée du Parser.

L'interface entre l'analyseur lexical et l'analyseur syntaxique doit être une fonction (par exemple *int nexSym()*), qui renvoie à chaque appel l'unité lexicale (symbole) suivante trouvée dans le texte source.

nextSym()

```
Token sym ;
void nextSym ()
    "ignore les espaces blancs et assigne le prochain lexème à sym"
      ...
}
```

Les deux analyseurs partagent la définitions de symboles.

```
interface Symbols {
    static final int ERROR = 0, EOF = ERROR + 1, IDENT = EOF + 1,
      LITERAL = IDENT + 1, LPAREN = LITERAL + 1, RPAREN = LPAREN + 1,
```

Analyseur lexical en dur (Exemple)

Soit la syntaxe des lexèmes en EBNF :

- symbol = {blank} (identifier | literal | "(" | ")" | "[" | "]" | "{" | "}" | "|" | "=" | ". ").
- Identifier = letter { letter | digit }.
- literal = "\"" {stringchar} "\"".
- stringchar = escapechar | plainchar.
- escapechar = "\\" char.
- plainchar = charNoQuote.

NB : Forme de Backus Naur étendue (EBNF)

Analyseur lexical en dur (suite)

Interface API de notre Analyseur lexical est de la forme suivante :

```
class Scanner mplements  Symbols {
  /** Constructor */
  Scanner (InputStream in)
  /** The symbol read = tokenclass last */
  int sym;
  /** The symbol's character representation */
   String chars;
  /** Read next token into sym and chars */
  void nextSym()
  /** Close input stream */
   void close()
}
```

Analyseur lexical en dur (suite)

```
import java.io.*;
class Scanner implements  Symbols {
public int sym;
public String chars;
/** the character stream being tokenized */
private InputStream in;
/** the next unconsumed character */
private char ch;
/** a buffer for assembling strings */
private StringBuffer buf = new StringBuffer();
/** the end of file character */
private final char eofCh = (char) -1;
public Scanner(InputStream in) { this.in = in; }
public static void error(String msg) {
    System.out.println("**** error: " + msg);
    System.exit(-1);
}
/** print current character and read next character */
```

Analyseur lexical en dur (suite)

```
  ...
/** print current character and read next character */
 rivate void nextCh() {
  System.out.print(ch);
  try {
    ch = (char) in.read();
   } catch (IOException ex) {
      error("read failure: " + ex.toString());
   }
 }
/** read next symbol */
...
 }
```

Analyseur lexical en dur (suite)

```
/** read next symbol */
 public void nextSym() {
   while (ch <= ' ') nextCh();
       switch (ch) {
        case 'a':   case 'b': ...  case 'z':
        case 'A':   case 'B':  ...  case 'Z':
            buf.setLength(0);  buf.append(ch);
            nextCh();
            while ('a' <= ch && ch <= 'z' ||
                   'A' <= ch && ch <= 'Z' ||
                   '0' <= ch && ch <= '9'){
                    buf.append(ch); nextCh();
           }
           sym = IDENT;
           chars = buf.toString(); break;


 }
```

Analyseur lexical en dur (suite)

```
    case '\"': nextCh();  buf.setLength(0);
            while (' ' <= ch && ch != eofCh && ch != '\"') {
              if (ch == '\\') nextCh();
              buf.append(ch); nextCh();
             }
            if (ch == '\"') nextCh();
            else error("unclosed string literal");
            sym = LITERAL; chars = buf.toString(); break;
      case '(': sym = LPAREN; nextCh(); break;
      case '[': sym = LBRACK; nextCh(); break;
      case '{': sym = LBRACE; nextCh(); break;
  ...
      case '|': sym = BAR; nextCh(); break;
      case '=': sym = EQL; nextCh(); break;
      case '.': sym = PERIOD; nextCh(); break;
      case eofCh: sym = EOF; break;
      default: error("illegal character: " + ch + "(" + (int)
    }
```

Analyseur lexical en dur (suite)

```
/** the string representation of a symbol */
 public static String representation(int sym) {
  switch (sym) {
   case ERROR:    return "<error>";
   case EOF:      return "<eof>";
   case IDENT:    return "identifier";
   case LITERAL:  return "literal";
   case LPAREN:   return "'('";
   case RPAREN:   return "')'";
   case EQL:   return "'='";
   case PERIOD :  return ".";
   default:       return "<unknown>";
   }
  }

 public void close() throws IOException {
   in.close(); }
```

Programme de teste

```java
class ScannerTest implements Symbols {
 static public void main(String[] args) {
    try {
        Scanner s = new Scanner(
                        new FileInputStream(args[0]));
        s.nextSym();
        while (s.sym != EOF) {
           System.out.println("[" +
              Scanner.representation(s.sym) + "]");
           s.nextSym();
       }
       s.close();
 }catch (IOException ex) {
    ex.printStackTrace();
    System.exit(-1);
 }  } }
```

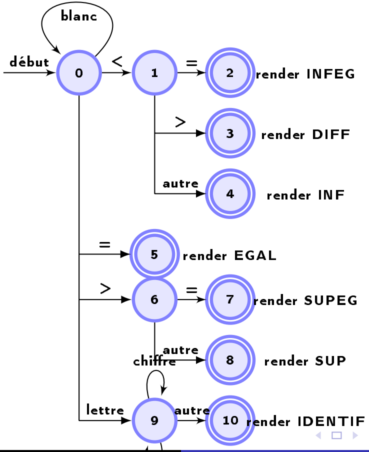
Analyseur lexical à l'aide d'automates (1)

Il suffira d'implémenter la fonction de transition de l'automate reconnaissant le langage.

	''	'\t'	'\n'	'<'	'='	'>'	lettre	chiffre	autre
0	0	0	0	1	5	6	9	erreur	erreur
1	4*	4*	4*	4*	2	3	4*	4*	4*
6	8*	8*	8*	8*	7	8*	8*	8*	8*
9	10*	10*	10*	10*	10*	10*	9	9	10

Analyseur lexical à l'aide d'automates

Diagrame de transition pour les opérateurs de comparaison et les identificateurs



Analyseur lexical à l'aide d'automates

Dans cet exemple, on peut obtenir un analyseur qui peut être plus rapide que celui de la première manière puisque l'essentiel du travail de l'analyseur se réduira à répéter bêtement l'action état = transit[état][nextCh( )] jusqu'à tomber sur un état final.
Un tableau identifié par *terminal* et indexé par les états, est défini par :

- terminal[e] = 0 si e n'est pas un état final (vu comme un boolean, terminal[e] est faux)
- terminal[e] = U + 1 si e est final, sans étoile et associé à l'unité lexicale U (en tant que booléen, terminal[e] est vrai, car les unités lexicales sont numérotées au moins à partir de zéro),
- terminal[e] = -(U + 1) si e est final, étoilé et associé à l'unité lexicale U (en tant que booléen, terminal[e] est encore vrai).
- Nous avons ajouté un état supplémentaire, ayant le numéro NBR_ETATS, qui correspond à la mise en erreur de l'analyseur lexical, et une unité lexicale ERREUR pour signaler cela.

Programme Scanner à l'aide d'automates

```
class ScannerAuomata{
  static final int NBR_ETATS = 10
  static final int  NBR_CARS = 256
  int transit[NBR_ETATS][NBR_CARS];
  int terminal[NBR_ETATS + 1];
  public ScannerAuomata(){
     transit = new int[NBR_ETATS][NBR_CARS];
     terminal = new int[NBR_ETATS+1];
     int i,j;
     for (i = 0; i < NBR_ETATS; i++)
       terminal[i] = 0;

     terminal[2] = INFEG + 1; terminal[3] = DIFF + 1;
     terminal[4] = - (INF + 1);terminal[5] = EGAL + 1;
     terminal[7] = SUPEG + 1; terminal[8] = - (SUP + 1);
     terminal[10] = - (IDENTIF + 1);
     terminal[NBR_ETATS] = ERREUR + 1;
```

Scanner à l'aide d'automates (suite)

```
for (i = 0; i < NBR_ETATS; i++)
      for (j = 0; j < NBR_CARS; j++)
           transit[i][j] = NBR_ETATS;

   transit[0][' '] = 0; transit[0]['\t'] = 0;
   transit[0]['\n'] = 0;  transit[0]['<'] = 1;
   transit[0]['='] = 5; transit[0]['>'] = 6;

   for (j = 'A'; j <= 'Z'; j++)
      transit[0][j] = 9;
   for (j = 'a'; j <= 'z'; j++)
      transit[0][j] = 9;
   for (j = 0; j < NBR_CARS; j++)
      transit[1][j] = 4;
   transit[1]['='] = 2; transit[1]['>'] = 3;
   for (j = 0; j < NBR_CARS; j++)
       transit[6][j] = 8;
   transit[6]['='] = 7;
```

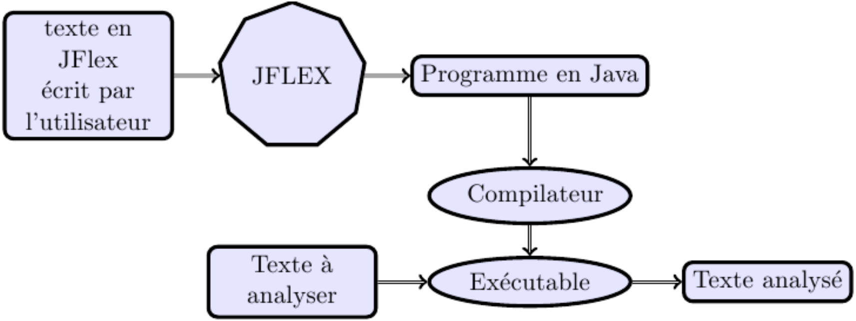
Scanner à l'aide d'automates (suite)

```
for (j = 0; j < NBR_CARS; j++)  transit[9][j] =10;
   for (j = 'A'; j <= 'Z'; j++)  transit[9][j] =9;
   for (j = 'a'; j <= 'z'; j++)  transit[9][j] =9;
   for (j = 0; j < 9; j++)  transit[9][j] =9;
}

int nextSym() {
   char ch;
   int etat = etatInitial;
   while(!terminal[etat]){
     ch = nextCh();
     etat = transit[etat][ch];
    }
   if (terminal[etat] < 0)
      ch = ' ' ; //blanc
   return abs(terminal[etat]) - 1;
 }
}//end
```

JFlex un générateur d'analyseur lexicaux

Principe général de génération d'analyseur lexical par JFlex



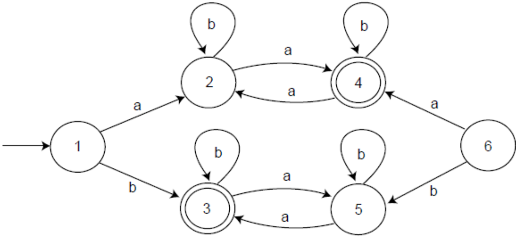
JFlex un générateur d'analyseur lexicaux

Minimisation d'un AFD
États inaccessibles (1)

- L'état 6 est inaccessible à partir de l'état initial.
- Si on enlève cet état, on obtient l'automate M_2 de la figure suivante

Minimisation d'un AFD
États inaccessibles (1)

Soit l'AFD $M_1 = \{1, 2, 3, 4, 5\}; \{a, b\}; 1; \{1, 34\}; \delta\}$

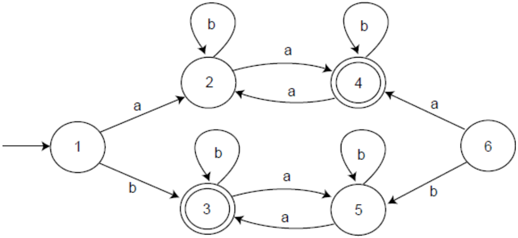


- L'état 6 est inaccessible à partir de l'état initial.
- Si on enlève cet état, on obtient l'automate M_2 de la figure suivante

Minimisation d'un AFD

États inaccessibles (1)

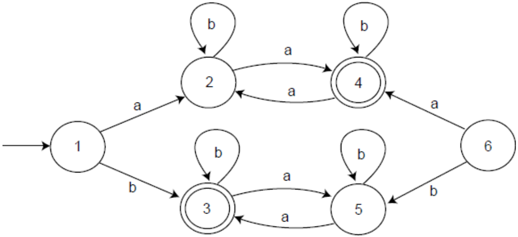
Soit l'AFD $M_1 = \{1, 2, 3, 4, 5\}; \{a, b\}; 1; \{1, 34\}; \delta\}$



- L'état 6 est inaccessible à partir de l'état initial.
- Si on enlève cet état, on obtient l'automate M_2 de la figure suivante

Minimisation d'un AFD
États inaccessibles (1)

Soit l'AFD $M_1 = \{1, 2, 3, 4, 5\}; \{a, b\}; 1; \{1, 34\}; \delta\}$

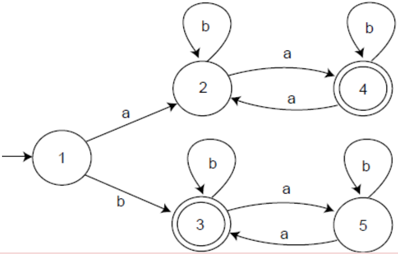


- L'état 6 est inaccessible à partir de l'état initial.
- Si on enlève cet état, on obtient l'automate M_2 de la figure suivante

Minimisation d'un AFD

États inaccessibles (2)

L'AFD M_2 obtenu en enlevant les états inaccessibles de M_1



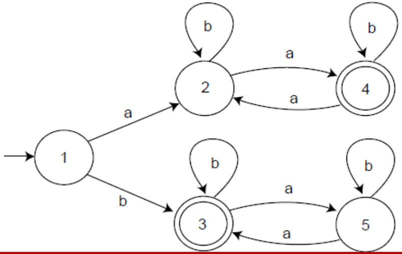
Regle 1

Soit M un AFD et soit M_0 l'AFD obtenu en enlevant tous les états inaccessibles de M. Alors L(M) = L(M_0).

Peut-on encore reduire le nombre d'états de M_2 ?

Minimisation d'un AFD
États inaccessibles (2)

L'AFD M_2 obtenu en enlevant les états inaccessibles de M_1



Regle 1

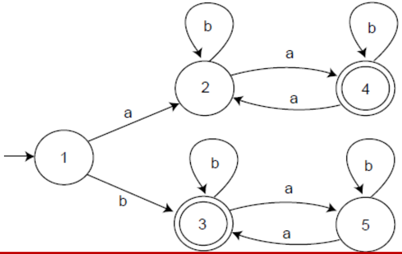
Soit M un AFD et soit M_0 l'AFD obtenu en enlevant tous les états inaccessibles de M. Alors $L(M) = L(M_0)$.

Peut-on encore reduire le nombre d'états de M_2 ?

Minimisation d'un AFD

États inaccessibles (2)

L'AFD M_2 obtenu en enlevant les états inaccessibles de M_1



Regle 1

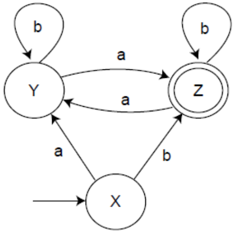
Soit M un AFD et soit M_0 l'AFD obtenu en enlevant tous les états inaccessibles de M. Alors $L(M) = L(M_0)$.

Peut-on encore reduire le nombre d'états de M_2 ?

Minimisation d'un AFD
L'AFD minimal

La réponse est encore oui !

L'AFD minimal M_3 *équivalent a* M_1



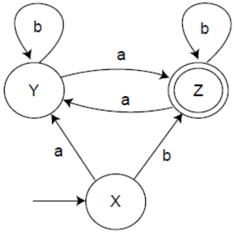
Cet automate est le plus petit AFD reconnaissant L(M1).

Comment a-t-on obtenu cet automate ?

Minimisation d'un AFD
L'AFD minimal

La réponse est encore oui !

L'AFD minimal M_3 *équivalent a* M_1



Cet automate est le plus petit AFD reconnaissant L(M1).

Comment a-t-on obtenu cet automate ?

Minimisation d'un AFD

Dénotons par $L_i (i = 1, 2, 3, 4, 5)$ l'ensemble des mots menant a un état final a partir de l'état i. Plus formellement, définissons :
$L_i = \{w \in \{a, b\}^* | \delta(i, w) \in F\}$

- $L_2 = L_5 = \{w \in \{a, b\}$ w contient un nombre impair de a $\}$
- $L_3 = L_4 = \{w \in \{a, b\}$ w contient un nombre pair de a $\}$

À chaque fois que l'on se retrouve dans l'état 2, on pourrait poursuivre l'éxecution à partir de l'état 5 sans rien changer au resultat.

Définition (états équivalents)

Deux états i et j sont equivalents si $L_i = L_j$. En d'autres termes, i et j sont equivalents si pour tout $w \in A$ on a :
$\delta(i, w) \in F$ si et seulement si $\delta(j, w) \in F$

Minimisation d'un AFD

Dénotons par $L_i (i = 1, 2, 3, 4, 5)$ l'ensemble des mots menant a un état final a partir de l'état i. Plus formellement, définissons :
$L_i = \{w \in \{a, b\}^* | \delta(i, w) \in F\}$

- $L_2 = L_5 = \{w \in \{a, b\}$ w contient un nombre impair de a $\}$
- $L_3 = L_4 = \{w \in \{a, b\}$ w contient un nombre pair de a $\}$

À chaque fois que l'on se retrouve dans l'état 2, on pourrait poursuivre l'exécution à partir de l'état 5 sans rien changer au resultat.

Définition (états équivalents)

Deux états i et j sont equivalents si $L_i = L_j$. En d'autres termes, i et j sont equivalents si pour tout $w \in A$ on a :
$\delta(i, w) \in F$ si et seulement si $\delta(j, w) \in F$

Minimisation d'un AFD

Dénotons par $L_i (i = 1, 2, 3, 4, 5)$ l'ensemble des mots menant a un état final a partir de l'état i. Plus formellement, définissons :
$L_i = \{w \in \{a, b\}^* | \delta(i, w) \in F\}$

- $L_2 = L_5 = \{w \in \{a, b\}$ w contient un nombre impair de a $\}$
- $L_3 = L_4 = \{w \in \{a, b\}$ w contient un nombre pair de a $\}$

À chaque fois que l'on se retrouve dans l'état 2, on pourrait poursuivre l'éxecution à partir de l'état 5 sans rien changer au resultat.

Définition (états équivalents)

Deux états i et j sont equivalents si $L_i = L_j$. En d'autres termes, i et j sont equivalents si pour tout $w \in A$ on a :
$\delta(i, w) \in F$ si et seulement si $\delta(j, w) \in F$

Minimisation d'un AFD

Dénotons par $L_i (i = 1, 2, 3, 4, 5)$ l'ensemble des mots menant a un état final a partir de l'état i. Plus formellement, définissons :
$L_i = \{w \in \{a, b\}^* | \delta(i, w) \in F\}$

- $L_2 = L_5 = \{w \in \{a, b\}$ w contient un nombre impair de a $\}$
- $L_3 = L_4 = \{w \in \{a, b\}$ w contient un nombre pair de a $\}$

À chaque fois que l'on se retrouve dans l'état 2, on pourrait poursuivre l'éxecution à partir de l'état 5 sans rien changer au resultat.

Définition (états équivalents)

Deux états i et j sont equivalents si $L_i = L_j$. En d'autres termes, i et j sont equivalents si pour tout $w \in A$ on a :
$\delta(i, w) \in F$ si et seulement si $\delta(j, w) \in F$

Minimisation d'un AFD

Dénotons par $L_i (i = 1, 2, 3, 4, 5)$ l'ensemble des mots menant a un état final a partir de l'état i. Plus formellement, définissons :
$L_i = \{w \in \{a, b\}^* | \delta(i, w) \in F\}$

- $L_2 = L_5 = \{w \in \{a, b\}$ w contient un nombre impair de a $\}$
- $L_3 = L_4 = \{w \in \{a, b\}$ w contient un nombre pair de a $\}$

À chaque fois que l'on se retrouve dans l'état 2, on pourrait poursuivre l'éxecution à partir de l'état 5 sans rien changer au resultat.

Définition (États équivalents)

Deux états i et j sont equivalents si $L_i = L_j$. En d'autres termes, i et j sont equivalents si pour tout $w \in A$ on a :
$\delta(i, w) \in F$ si et seulement si $\delta(j, w) \in F$

Minimisation d'un AFD
Etats équivalents

Ainsi, dans l'AFD M_2 les états 2 et 5 sont équivalents et les états 3 et 4 sont équivalents. L'etat 1 n'est équivalent a aucun autre état.

Remarque

si i et j sont deux états équivalents d'un automate donne, alors a chaque fois que l'on se retrouve dans l'état i, on pourrait poursuivre l'éxecution à partir de l'état j sans rien changer au resultat. Cela conduit a l'observation suivante :

Observation

Il est toujours possible d'ajouter des $\varepsilon-$transitions entre deux états équivalents sans rien changer au langage reconnu.

Minimisation d'un AFD

Etats équivalents

Ainsi, dans l'AFD M_2 les états 2 et 5 sont équivalents et les états 3 et 4 sont équivalents. L'etat 1 n'est équivalent a aucun autre état.

Remarque

si i et j sont deux états équivalents d'un automate donne, alors a chaque fois que l'on se retrouve dans l'état i, on pourrait poursuivre l'éxecution à partir de l'état j sans rien changer au resultat. Cela conduit a l'observation suivante :

Observation

Il est toujours possible d'ajouter des $\varepsilon-$transitions entre deux états équivalents sans rien changer au langage reconnu.

Minimisation d'un AFD

Etats équivalents

Ainsi, dans l'AFD M_2 les états 2 et 5 sont équivalents et les états 3 et 4 sont équivalents. L'etat 1 n'est équivalent a aucun autre état.

Remarque

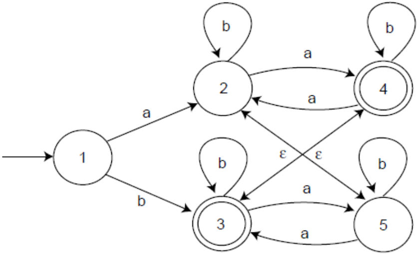
si i et j sont deux états équivalents d'un automate donne, alors a chaque fois que l'on se retrouve dans l'état i, on pourrait poursuivre l'éxecution à partir de l'état j sans rien changer au resultat. Cela conduit a l'observation suivante :

Observation

Il est toujours possible d'ajouter des $\varepsilon-$transitions entre deux états équivalents sans rien changer au langage reconnu.

Minimisation d'un AFD

L'AFN M_4 obtenu a partir de M_2 en ajoutant des $\varepsilon-$transitions entre les paires d'états équivalents.

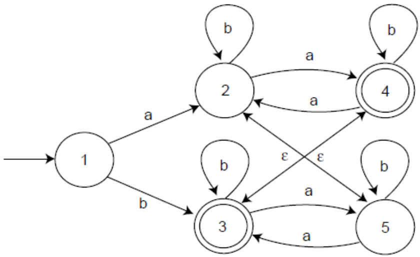


Remarque

L'automate obtenu n'est plus deterministe. Cependant, on peut obtenir un AFD M_5 a partir de L'AFN M_4 en utilisant la methode de détermination.

Minimisation d'un AFD

L'AFN M_4 obtenu a partir de M_2 en ajoutant des $\varepsilon-$transitions entre les paires d'états équivalents.

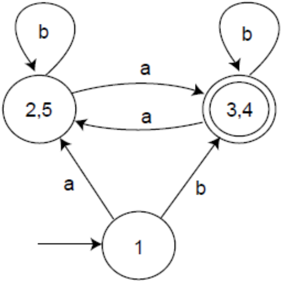


Remarque

L'automate obtenu n'est plus deterministe. Cependant, on peut obtenir un AFD M_5 a partir de L'AFN M_4 en utilisant la methode de détermination.

Minimisation d'un AFD

L'AFD M_5 obtenu a partir de L'AFN M_4



La suite

Analyse Santaxique

Lectures complémentaires I

- Modern compilers in C/Java/ML (3 livres jumeaux). de A. Appel. Cambridge University Press, 1998.
- Compilateurs : Principes, techniques et outils. de A. Aho, R. Sethi, and J. Ullman. Dunod, 2000.
- Programming Language Processors - Compilers and Interpreters de David Watt, Prentice-Hall International Series in Computer Science.