Université Mohammed V - Agdal
Faculté des sciences
Département d'Informatique

Cours de Compilation

SMI - S5

Prof. M.D. RAHMANI
mrahmani@fsr.ac.ma

III- L'analyse lexicale - Plan:

1- Le rôle d'un analyseur lexical

2- Terminologie

3- Spécification des unités lexicales

- 3.1- Chaînes et langages
- 3.2- Opérations sur les langages
- 3.3- Expressions régulières
- 3.4- Définitions régulières

4- Reconnaissance des unités lexicales

5- Le langage FLEX

- 5.1- Structure d'un programme FLEX
- 5.2- Ecriture d'un analyseur lexical avec FLEX

III- L'analyse lexicale

6- Automates à états finis (AEF)

- 6.1- Automates à états finis non déterministes (AFN)
- 6.2- Tables de transition
- 6.3- Automates à états finis déterministes (AFD)

7- Grammaires régulières

8- Des expressions régulières aux automates

- 8.1- Conversion d'un AFN en AFD
- 8.2- Construction d'un AFN à partir d'une expression régulière
- 8.3- Minimisation du nombre d'états d'un AFD

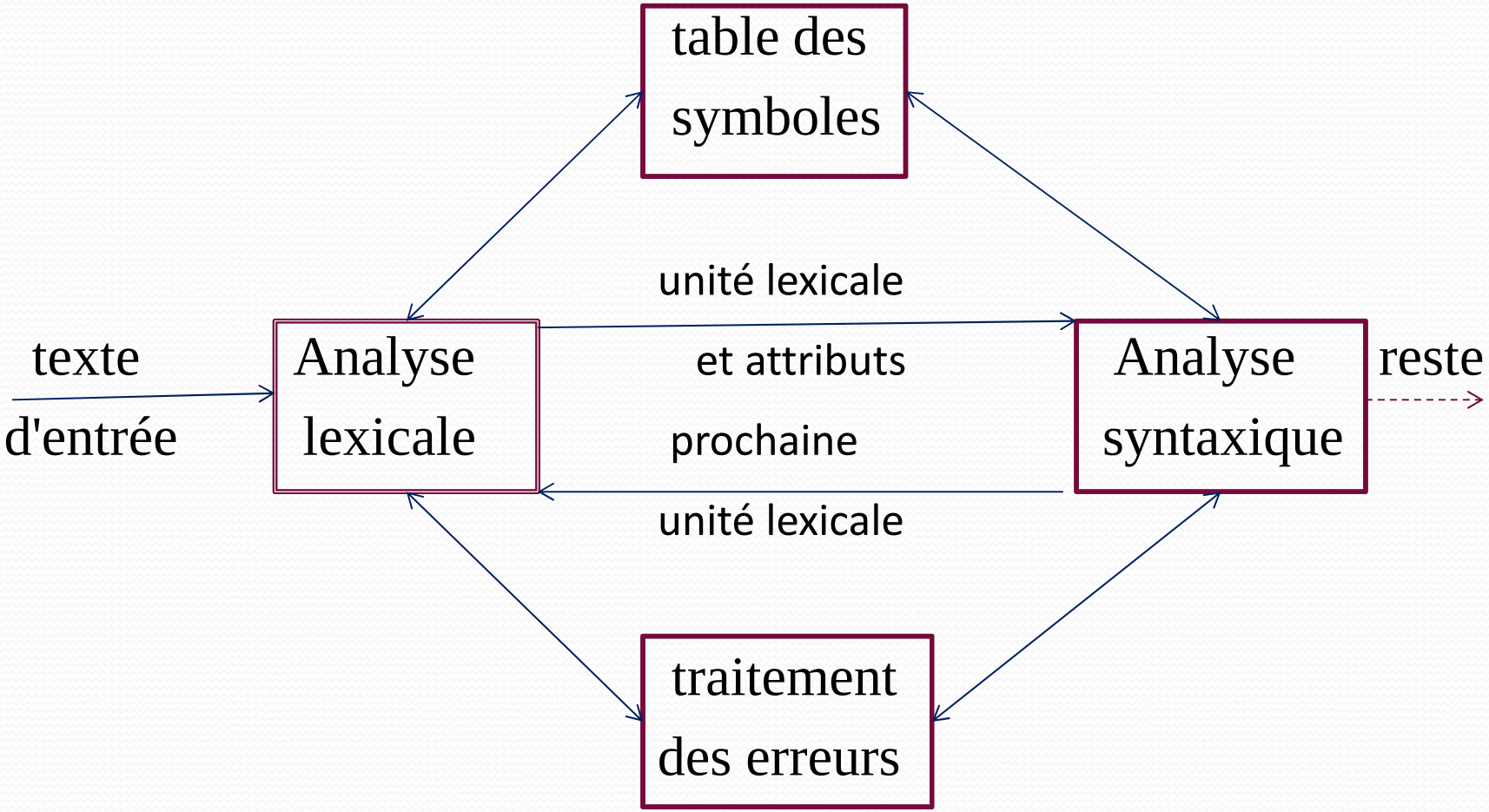
1- Le rôle d'un analyseur lexical:

L'analyseur lexical est chargé de lire le texte d'entrée, caractère par caractère, de la gauche vers la droite et isoler les mots et leur classe.

De plus, il doit:

- éliminer les blancs (espaces, tabulations, fin de lignes) et les commentaires.
- détecter les erreurs et associer des messages d'erreurs.

1- Le rôle d'un analyseur lexical:



Interaction entre analyseur lexical et analyseur syntaxique.

2- Terminologie

- **Unité lexicale:** est un symbole terminal de la grammaire du langage.

- **Modèle:** est une règle qui décrit un ensemble de chaînes associées à la même unité lexicale.

- **lexème:** est une suite de caractères du texte d'entrée qui concorde avec le modèle.

Exemple: **35** est un **lexème** (un mot) qui appartient à l'**unité lexicale** (la classe) **nombre**.

2- Terminologie

Remarques:

Dans de nombreux langages, les classes suivantes couvrent la plupart des unités lexicales:

1- Une unité lexicale pour chaque mot clé.

2- Des unités lexicales pour les opérateurs, soit individuellement, soit par classes.

3- Une unité lexicale pour les identificateurs (noms de variables, fonctions, tableaux, structures...).

4- Une ou plusieurs unités lexicales pour les nombres et les chaînes.

5- Une unité lexicale pour chacun des signes de ponctuation, tels que les parenthèses gauche et droite, la virgule , le point-virgule...

3- Spécification des unités lexicales

3.1- Chaînes et langages:

Définitions générales:

- *Un alphabet* Σ *ou une classe de caractères définit un ensemble fini de symboles.*

Exemples: {0,1} : l'alphabet binaire

ASCII : l'alphabet informatique

- *Une chaîne ou un mot sur un alphabet* Σ *est une séquence finie de symboles extraits de cet ensemble.*

3- Spécification des unités lexicales

- *La longueur d'une chaîne s est notée: $|s|$*

  La longueur de la chaîne vide notée ε : $|\varepsilon| = 0$

- *L'ensemble des mots sur l'alphabet Σ est noté Σ^**

- *Un mot $u \in \Sigma^*$ est <u>facteur</u> du mot $w \in \Sigma^*$ s'il existe v, $v' \in \Sigma^*$ tels que: $w = v\,u\,v'$.*

- *Un mot fini u est <u>périodique</u> si $u = x^n$ pour $n \geq 2$.*

  Tout mot non périodique est dit <u>primitif</u>.

3- Spécification des unités lexicales

Soit une chaîne ***s*** :

- **préfixe de *s*** *est une chaîne obtenue en supprimant un nombre quelconque (même nul) de symboles à la fin de* **s**.
- **suffixe de *s*** *est une chaîne obtenue en supprimant un nombre quelconque (même nul) de symboles au début de* **s**.
- **sous-chaîne de *s*** *est une chaîne obtenue en supprimant un préfixe et un suffixe.*
- **sous-suite de s** *est une chaîne obtenue en supprimant un nombre quelconque (même nul) de symboles non nécessairement consécutifs.*
- **Un langage** *est un ensemble quelconque de chaînes construites sur un alphabet fixé.*

3- Spécification des unités lexicales

3.2- Opérations sur les langages:

Soit L et M deux langages:

- Union de L et M : $L \cup M = \{ \forall s / s \in L \text{ ou } s \in M\}$

- Concaténation de L et M : $LM = \{ st / s \in L \text{ et } t \in M\}$

- Fermeture de Kleene de L : $L^* = \bigcup_{i=0}^{\infty} L^i$

L^* dénote un nombre quelconque (même nul) de concaténation de L.

On note $L^0 = \{\varepsilon\}$

- Fermeture positive de L : $L^+ = \bigcup_{i=1}^{\infty} L^i$

3- Spécification des unités lexicales

Exemples:

Soit $L = \{A,B,...,Z\} \cup \{a,b,...,z\}$

et $C = \{0,1,...9\}$

A partir de *L* et *C*, nous pouvons produire d'autres langages.

- ***L U C*** *:* ensemble des lettres et chiffres,
- ***LC*** *:* ensemble des chaînes constituées d'une lettre suivie d'un chiffre,
- L^4 *:* ensemble des chaînes constituées de 4 lettres,
- C^+ *:* ensemble des entiers naturels,
- $L(LUC)^*$ *:* ensemble des chaînes constituées d'une lettre suivie d'une chaîne de lettres et de chiffres ou d'une chaîne vide.

3- Spécification des unités lexicales

3.3- Expressions régulières:

Une expression régulière est une notation qui permet de décrire un ensemble (une classe) de chaînes de caractères.

Exemple: Un nombre entier non signé est une chaîne constituée d'une suite de chiffres, au moins un.

L'expression régulière associée est: **`(chiffre)+`**

+ *est un opérateur unaire post-fixe qui veut dire un ou plusieurs fois.*

3- Spécification des unités lexicales

Les règles qui définissent les expressions régulières sur un **alphabet Σ** sont:

- **ε** est une expression régulière qui dénote **{ε}** c-à-d l'ensemble dont le seul élément est la **chaîne vide ε**.

- si **a** est un symbole de l'alphabet **Σ**, alors **a** est une expression régulière qui dénote **{a}**, c-à-d l'ensemble constitué de la chaîne **a**.

3- Spécification des unités lexicales

- soit ***r*** et ***s*** deux expressions régulières qui dénotent les langages L(r) et L(s), alors:
  - **(r) | (s)** est une expression régulière qui dénote **(L(r)) U (L(s))**.
  - **(r)(s)** est une expression régulière qui dénote **(L(r))(L(s))**.
  - **(r)*** est une expression régulière qui dénote **(L(r))***.
    - *Les langages dénotés par les expressions régulières sont appelés* ***langages réguliers***.

3- Spécification des unités lexicales

Exemples:

a|b*c : les chaînes constituées, soit d'un **a**, ou d'un nombre quelconque, éventuellement nul, de la lettre **b** suivie de la lettre **c**.

a | b = {a,b}

(a|b)(a|b) = {aa,ab,ba,bb}

Définition:

Si deux expressions **r** et **s** dénotent le même langage, on dit qu'elles sont équivalentes et on écrit: **r=s**

Exemple: (a|b)= (b|a)

3- Spécification des unités lexicales

Propriétés algébriques sur les expressions régulières:

Soit **r,s** et **t** des expressions régulières.

r|s = s|r : *l'opérateur* **|** *(ou) est commutatif.*

r|(s|t) = (r|s)|t : *l'opérateur* **|** *est associatif.*

(rs)t = r(st) : *la concaténation est associative.*

r(s|t)=rs|rt : *la concaténation est distributive par rapport au* **|**

ε r = r ε = r : *ε est l'élément neutre de la concaténation.*

r* =(r|ε)* : *ε est inclus dans une fermeture.*

r = r*** : ***** *est idempotent*

Remarque: la chaîne vide **ε = s^0**

3- Spécification des unités lexicales

Notations:

***** est un opérateur unaire poste-fixe qui veut dire zéro, un ou plusieurs fois.

+ est un opérateur unaire poste-fixe qui veut dire un ou plusieurs fois.

r+ = r r* = r*r

r* = r+|ε

? est un opérateur unaire poste-fixe qui veut dire zéro ou une fois.

r? = r|ε

[a-z] désigne un élément (une lettre) de cette classe.

[a-z] = a|b|c...|z

3- Spécification des unités lexicales

Conventions:

1- L' opérateur unaire poste-fixe ***** a la plus haute priorité et est associatif à gauche.

2- Les opérateurs **+** et **?** ont la même priorité et la même associativité que *****.

2- La concaténation a la deuxième priorité et est associative à gauche.

3- Le **|** a la plus faible priorité et est associatif à gauche.

Selon ces conventions, **(a)|((b)*(c))** est équivalente à **a|b*c**

3- Spécification des unités lexicales

Exemples d'expressions régulières:

- Un identificateur: **`lettre(lettre|chiffre)*`**
  `= [a-zA-Z][a-zA-Z0-9]*`

- Un entier signé ou non: **`(+|-)? (chiffre)+`**
  `= [+-]?[0-9]+`
- Un nombre décimal: **`(+|-)? (chiffre)+ (.(chiffre)+)?`**

- Un réel:

```
(+|-)?(chiffre)+(.(chiffre)+)?((e|E)(+|-)?(chiffre)+)?
 = [+-]?[0-9]+(.[0-9]+)?((e|E)(+|-)?[0-9]+)?
```

3- Spécification des unités lexicales

3.4- Définitions régulières:

Une définition régulière est une suite de définitions de la forme:

$d_1 \longrightarrow r_1$

$d_2 \longrightarrow r_2$

.

.

$d_n \longrightarrow r_n$

Chaque d_i est un nom distinct et chaque r_i est une expression régulière sur les symboles : $\Sigma \cup \{d_1, d_2, ..., d_{i-1}\}$

Nous allons voir quelques exemples de définitions régulières :

3- Spécification des unités lexicales

Exemples:

1- Définition régulière d'un identificateur:

```
lettre  ⟶  A|B|...|Z|a|b|...|z
chiffre ⟶  0|1|2...|9
id      ⟶  lettre(lettre|chiffre)*
```

2- Définition régulière des entiers signés et non signés:

```
chiffre ⟶  0|1|2...|9
entier  ⟶  [+|-]?(chiffre)+
```

3- Spécification des unités lexicales

3- Définition régulière d'un réel:

L'alphabet Σ={0,1,...,9,.,e,E,+,-}

Une définition régulière sera:

chiffre ⟶ 0|1|2...|9

chiffres ⟶ (chiffre)+

p_entiere ⟶ (+|-)? chiffres

p_decimale ⟶ (.chiffres)?

p-puissance ⟶ ((e|E)(+|-)? chiffres)?

reel ⟶ p_entiere p_decimale p_puissance

4- Reconnaissance des unités lexicales

Soit le fragment de grammaire des instructions conditionnelles:

```
inst  ⟶  si (exp) alors inst
         |si (exp) alors inst sinon inst
         |autre_inst
exp   ⟶  terme operel terme
         |terme
terme ⟶  id
         |nb
```

Les terminaux de cette grammaire sont:

`si`, `alors`, `sinon`, `(`, `)`, `operel`, `id` et `nb`.

Pour les reconnaître, nous allons d'abord donner les définitions régulières associées.

4- Reconnaissance des unités lexicales

4.1- Définitions régulières des terminaux de la grammaire:

A noter qu'il faut reconnaitre les blancs aussi pour les ignorer.

```
delim  ⟶ espace|tabulation|fin_de_ligne
blanc  ⟶ (delim)+
IF     ⟶ si
THEN   ⟶ alors
ELSE   ⟶ sinon
operel ⟶ <|<=|==|<>|>=|>
id     ⟶ [A-Za-z][A-Za-z0-9]*
nb     ⟶ (+|-)?[0-9]+(.[0-9])?((+|-)?(e|E)[0-9]+)?
```

Remarque: *Les commentaires et les blancs sont traités comme des modèles qui ne retournent aucune unité lexicale.*

4- Reconnaissance des unités lexicales

4.2- Diagramme de transition:

Un diagramme de transition est un organigramme orienté qui décrit les actions à réaliser par l'analyseur lexical.

Il est constitué d'états et de transition entre états, définis par les notations suivantes:


un état

un arc ou une transition

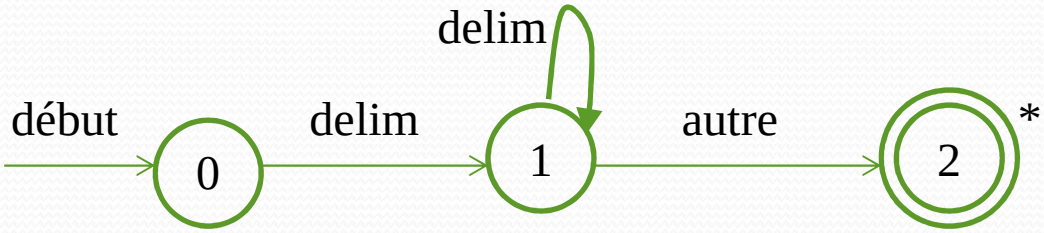

état d'acceptation, c-à-d que le lexème est reconnu.


* état d'acceptation avec recul

Remarque: *en pratique, une transition correspond à la consommation d'un caractère et un seul.*

4- Reconnaissance des unités lexicales

1- Diagramme de transition des `blancs`:



(ne rien retourner)

Remarque: ***`autre`*** veut dire, autre que les autres arcs sortants du même état.

Dans ce cas, ***`autre`*** veut dire autre qu'un délimiteur.

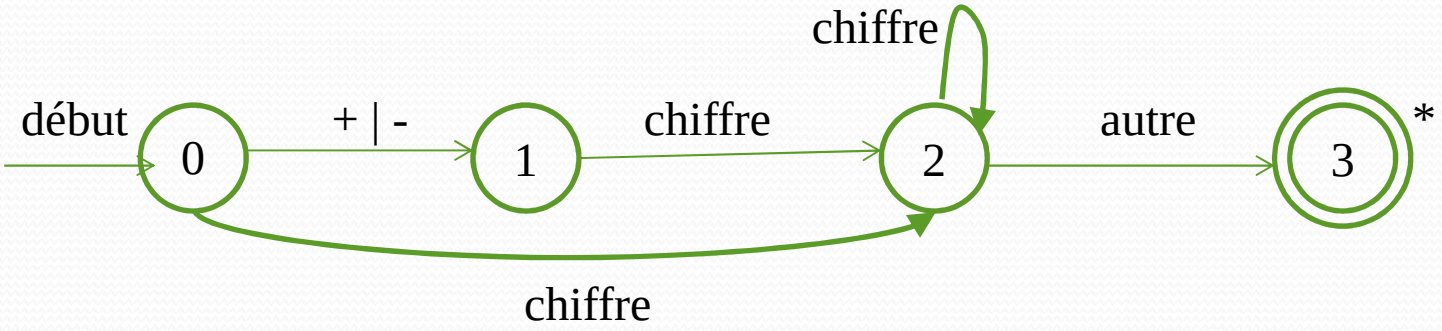
4- Reconnaissance des unités lexicales

2- Diagramme de transition des `identificateurs`:

Remarque: ***`autre`*** veut dire, autre que ***`lettre`*** et ***`chiffre`***.

4- Reconnaissance des unités lexicales

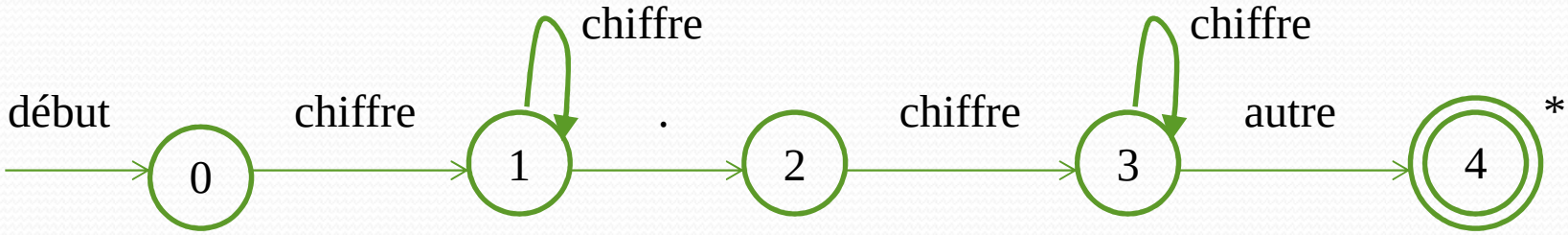
Diagramme de transition des `entiers signés ou non`:



Remarque: *`autre`* veut dire, **autre que les `chiffres`.**

4- Reconnaissance des unités lexicales

Diagramme de transition des nombres `décimaux non signés`:

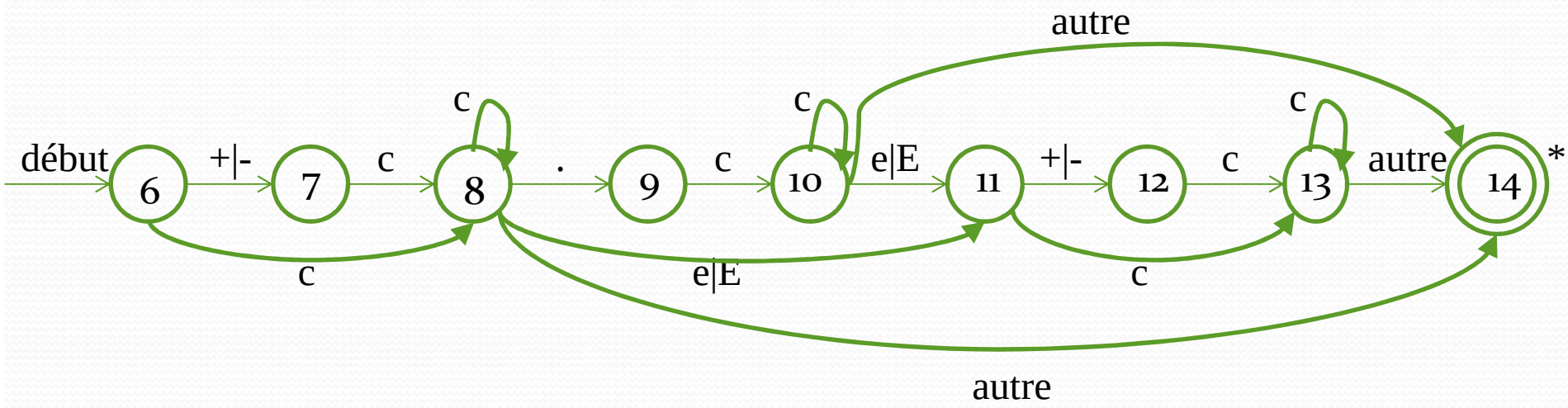


Expression régulière: `(chiffre)+.(chiffre)+`

Remarque: *Nous exigeons par ce diagramme au moins un chiffre après le point.*

4- Reconnaissance des unités lexicales

3- Diagramme de transition des `nombres réels`:

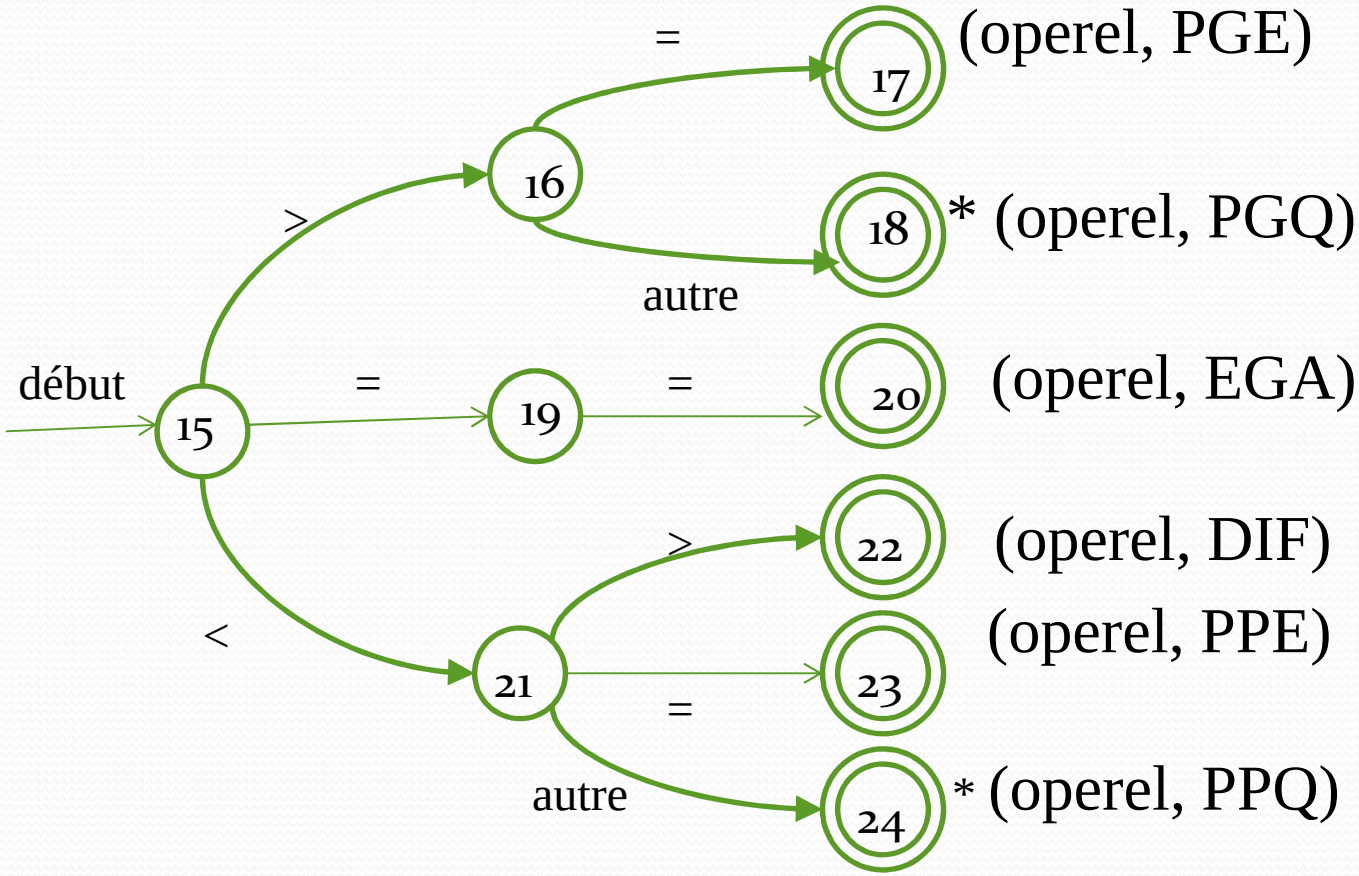


*A l'état **14** d'acceptation avec recul, nous retournons l'unité lexicale **nb** et un pointeur sur le lexème reconnu.*

4- Reconnaissance des unités lexicales

4- Diagramme de transition des `opérateurs de relation`

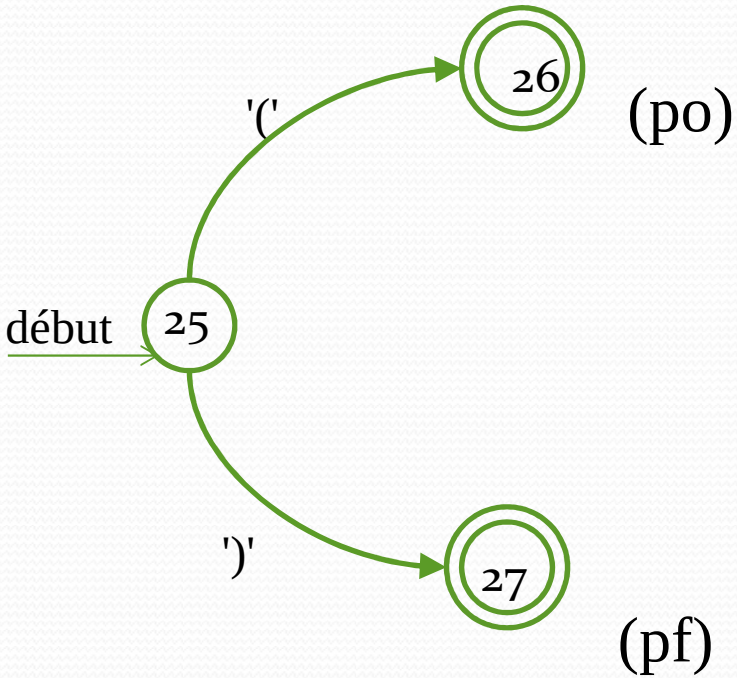
Expression régulière: <|<=|<>|>|>=|==|<>



4- Reconnaissance des unités lexicales

5- Diagramme de transition des `parenthèses`

Expression régulière: `'('|')'`



4- Reconnaissance des unités lexicales

4.2- Implantation des diagrammes de transition:

4- Reconnaissance des unités lexicales

4.3- La table de symboles:

La table des symboles est une structure de données constituée des champs suivants:

- un pointeur (`ptrlex`) pointant sur l'adresse de la 1ère occurence d'un lexème figurant dans le tampon (`lexèmes`);

- une chaîne de caractères (`unilex`) qui contient l'unité lexicale du lexème detecté;

- un attribut qui est un indice de la position du lexème dans la table des symboles.

4- Reconnaissance des unités lexicales

Supposons que le texte d'entrée est: **`"si gamma=10 alors aire >= 78 sinon g>1.3"`**

La table des symboles aura la forme suivante:

ptrlex	unilex	indice
0	si	1
3	id	2
9	operel	3
11	nb	4
14	alors	5
20	id	6
25	operel	7
28	nb	8
31	sinon	9
37	id	10
39	operel	11
41	nb	12

4- Reconnaissance des unités lexicales

La chaîne engendrée est:

```
"si$gamma$=$10$alors$aire$>=78$sinon$g$>$1.3"
```

Pour implanter la table des symboles, nous avons besoin des 2 fonctions suivantes:

- une fonction d'insertion;

- une fonction de recherche.

4- Reconnaissance des unités lexicales

L'algorithme de la fonction d'insertion:

```
Fonction inserer
début
  indice ←——— indice+1;
  TS[indice].ptrlex ←———
  TS[indice].unilex ←———
  retourner(indice)
fin inserer
```

création d'une nouvelle entrée de la TS

l'adresse du début du lexème dans le tampon `lexèmes`

l'unité lexicale associée

4- Reconnaissance des unités lexicales

L'algorithme de la fonction de recherche:

```
Fonction chercher
début
   j ←———— 0;  pour parcourir la TS
   trouve=faux;  boolean pour arrêter la recherche
   tant que (j<taille de la TS et non trouvé) faire
       si TS[j].unilex=l'UL du lexème à chercher alors
              si TS[j].ptrlex pointe sur le lexème
              alors
                     trouve ←——— vrai;
                     retourne(j);
              sinon j ←———  j+1;
       sinon trouve retourne(-1)
Fin Fonction chercher
```

4- Reconnaissance des unités lexicales

Un programme de la TS en langage C:

5- Le langage FLEX

Un programme écrit en langage ***LEX*** construit d'une manière automatique un analyseur lexical.

Il prend en entrée un ensemble d'*expressions régulières* et de *définitions régulières* et produit en sortie un *code cible en langage C*.

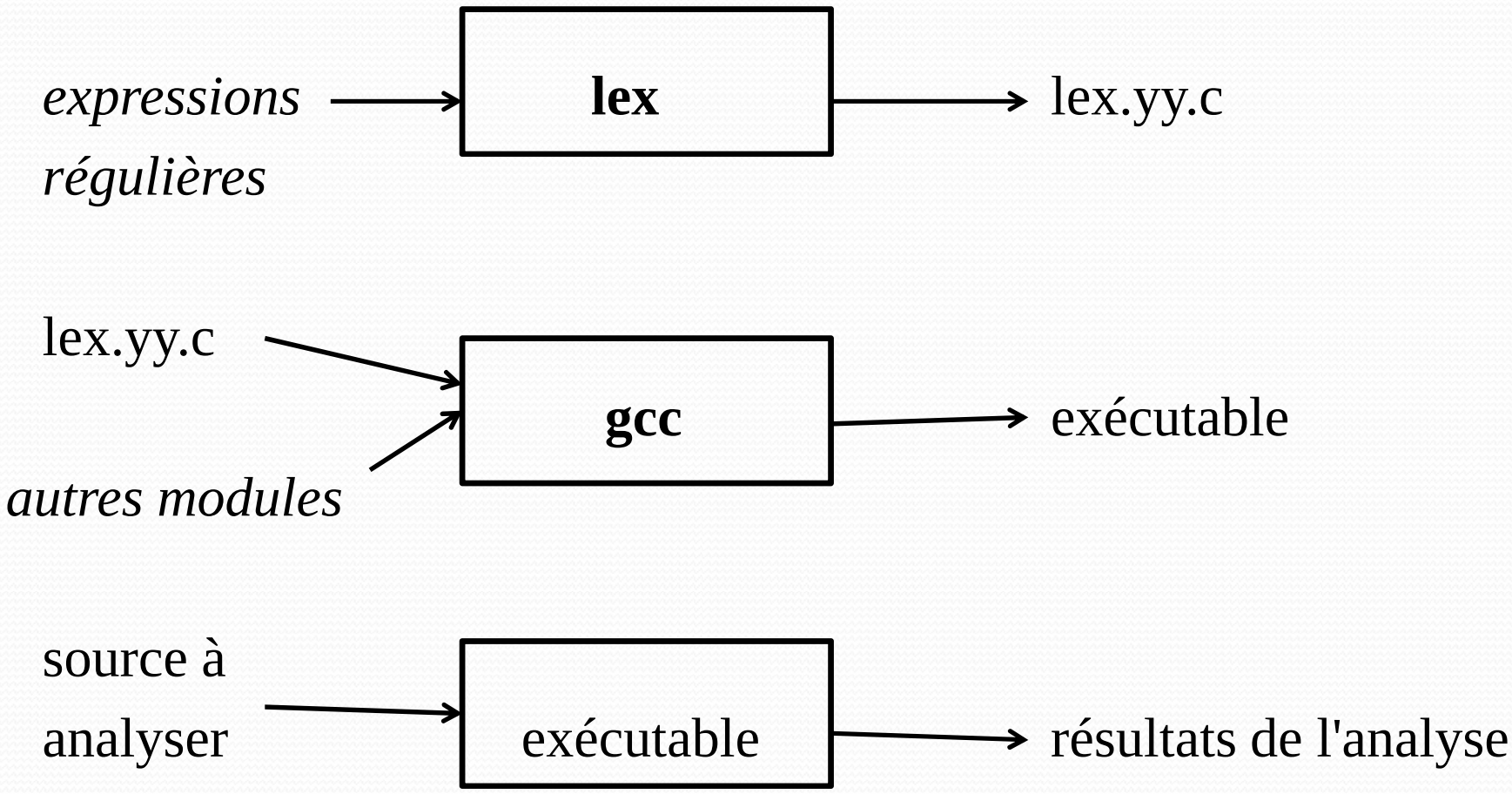
Ce code en langage C doit être compilé par le compilateur du langage C pour produire un exécutable qui est un analyseur lexical correspondant au langage défini par les expressions régulières d'entrée.

Plusieurs langages dérivés de LEX existent;

- Flex produit du C++
- Jlex produit du Java
- Lecl produit du Caml

5- Le langage FLEX

La compilation d'un code source en LEX se fait en deux étapes.



5- Le langage FLEX

5.1- Structure d'un programme FLEX:

Un programme source en langage lex est constitué de 3 sections délimitées par %%:

```
//1ère partie: déclarations
%{
  déclarations pour le compilateur C
%}
  définitions régulières
%%
// 2ème partie: règles de traduction
  expressions régulières + actions à réaliser
%%
// 3ème partie fonctions en C
  Fonctions annexes en langage C
```

5- Le langage FLEX

- La 1ère partie est constituée de:

  - déclarations en langage C des bibliothèques, variables, structures, unions...

  - déclarations de définitions régulières utilisables par les règles de traduction.

- La 2ème partie a la forme:

m_1	{action$_1$}	avec m_i une expression régulière ou une
m_2	{action$_2$}	définition régulière de la 1ère partie
...	....	et
m_i	{acion$_i$}	*action$_i$* est l'action à réaliser par l'analyseur
...	....	lexical si un lexème est accepté par m_i.
m_n	{action$_n$}	

5- Le langage FLEX

- La 3ème partie est une suite de fonctions en C qui aident à l'analyse dans l'analyse par les règles de traduction de la 2ème partie.

  Cette partie peut contenir une fonction *main* du langage C.

Remarques:

- Le code doit commencer à la 1ère colonne.
- La 1ère partie et la dernière sont facultatives.
- Le fichier LEX doit avoir l'extension ***.l***

Exemple du code en langage LEX: ***espace.l***

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

5- Le langage FLEX

5.2- Ecriture d'un analyseur lexical avec FLEX: *analex.l*

1ère partie:

```
//déclarations en C
%{
#include<stdio.h>
%}
delim  [ \t\n]
bl     {delim}+
lettre [a-zA-Z]
chiffre [0-9]
id  {lettre}({lettre}|{chiffre})*
nb  (\+|\-)?{chiffre}+(\.{chiffre}+)?((e|E)(\+|\-)?{chiffre}+)?
% %
```

Remarque: Les caractères **+**, **-** et **.** sont précédés de **** pour les distinguer des opérateurs correspondants.

5- Le langage FLEX

2ème partie:

```
{bl}   {/* supprimer de la sortie */}
sinon   {printf("\n Mot clé: ELSE\n");}
si      {printf("\n Mot clé: IF\n");}
alors   {printf("\n Mot clé: THEN\n");}
{id}    {printf("\n Identificateur:%s\n",yytext);}
{nb}    {printf("\n Nombre:%s\n",yytext);}
"<="    {printf("\n Operateur relationnel: PPE\n");}
"<>"    {printf("\n Operateur relationnel: DIF\n");}
"<"     {printf("\n Operateur relationnel: PPQ\n");}
">="    {printf("\n Operateur relationnel: PGE\n");}
"<"     {printf("\n Operateur relationnel: PGQ\n");}
"=="    {printf("\n Operateur relationnel: EGA\n");}
"("     {printf("\n PO\n");}
")"     {printf("\n PF\n");}
.       {printf("\n%s: Caractère non reconnu\n",yytext);}
%%
```

5- Le langage FLEX

3ème partie:

```
int main()
{
  FILE *fichier;
  printf("Nom du fichier à analyser");
  scanf("%s",&fichier);
  yyin=fichier;
  yylex();
  return 0;
}
```

5- Le langage FLEX

Remarques:

"." est un opérateur qui veut dire tous caractère sauf le retour à la ligne.

"^" est un opérateur pour le complémentaire d'une classe.

`yytext` est un pointeur sur la chaîne analysée.

`yyin` correspond à l'entrée

`yylex()` est la fonction principale du programme écrit en LEX.

Options de compilation:

Avec LEX: 1/ *lex analex.l* produit *lex.yy.c*

2/ *cc lex.yy.c -ll* pour *library lex*

Avec FLEX: 1/ *flex analex.l* produit lex.yy.c

2/ *gcc lex.yy.c -lfl* pour *library fast lex*

6- Automates à états finis (AEF)

Les automates à états finis sont des graphes orientés à l'image des diagrammes de transition, avec certaines différences:

1- Les automates à états finis sont des *reconnaisseurs*; ils disent simplement "oui" ou "non" à propos de chaque chaîne d'entrée.

2- Il y'a 2 types d'automates à états finis:

- Les automates à états finis non déterministes (AFN) n'ont aucune restrictions sur les étiquettes de leurs arcs.

  Un symbole peut étiqueter plusieurs arcs partant d'un même état, et la chaîne vide ε est une étiquette possible.
- Les automates à états finis déterministes (AFD), pour lesquels, ne peuvent pas partir plusieurs transitions du même état avec le même caractère et n'accepte pas d'ε-transition..

6- Automates à états finis (AEF)

6.1- Automates à états finis non déterministes (AFN):

Un AFN se compose de:

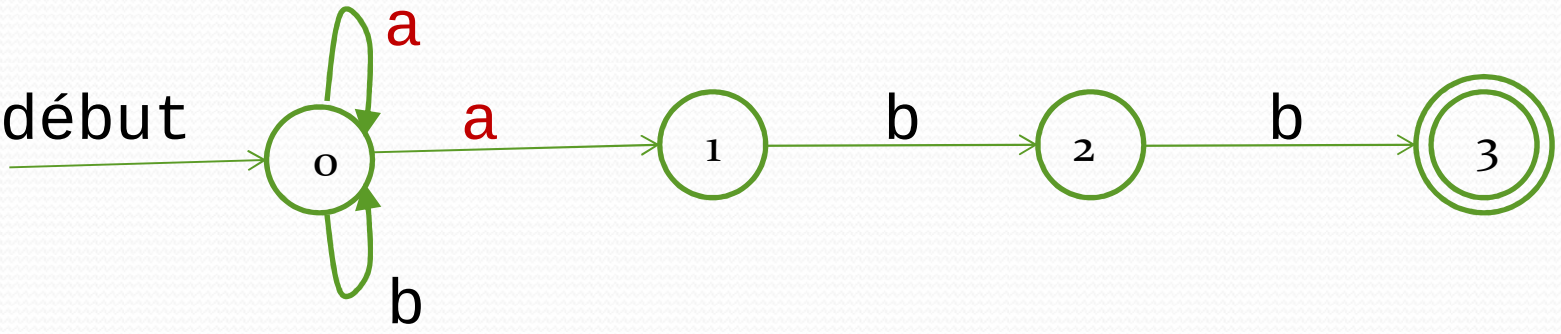
- Un ensemble fini S d'états.
- Un ensemble Σ de symboles d'entrée, l'alphabet du langage.

  On considère que la chaîne vide ε, n'est jamais un membre de Σ.
- Une fonction de transition qui donne pour chaque état et pour chaque symbole de Σ U {ε}, l'ensemble des états suivants.
- Un état s_0 appartenant à S, qui est l'état de départ.
- Un ensemble d'états F, sous-ensemble de S, l'ensemble des états d'acceptation ou états finaux.

6- Automates à états finis (AEF)

6.1- Automates à états finis non déterministes (AFN):

Exemple: L'AFN qui reconnait le langage défini par l'expression régulière : **(a|b)* abb**



Remarque: *Le non déterminisme ici est associé à deux arcs sortants de l'état 0 avec le même symbole a.*

6- Automates à états finis (AEF)

6.2- Tables de transition:

Nous pouvons représenter un AFN par une table de transition, dont les lignes correspondent aux états et les colonnes aux symboles d'entrée et à ε.

L'entrée pour un état donné et une entrée donnée est la valeur de la fonction de transition appliquée à ces arguments.

Exemple: soit l'AFN précédant

Symbole / Etat	a	b	ε
0	{0,1}	{0}	-
1	-	{2}	-
2	-	{3}	-
3	-	-	-

6- Automates à états finis (AEF)

6.3- Automates à états finis déterministes (AFD):

Un AFD est un cas particulier d'un AFN où:

- il n'y a aucun arc étiqueté par ε,
- pas plus d'un arc avec le même symbole sortant d'un état.

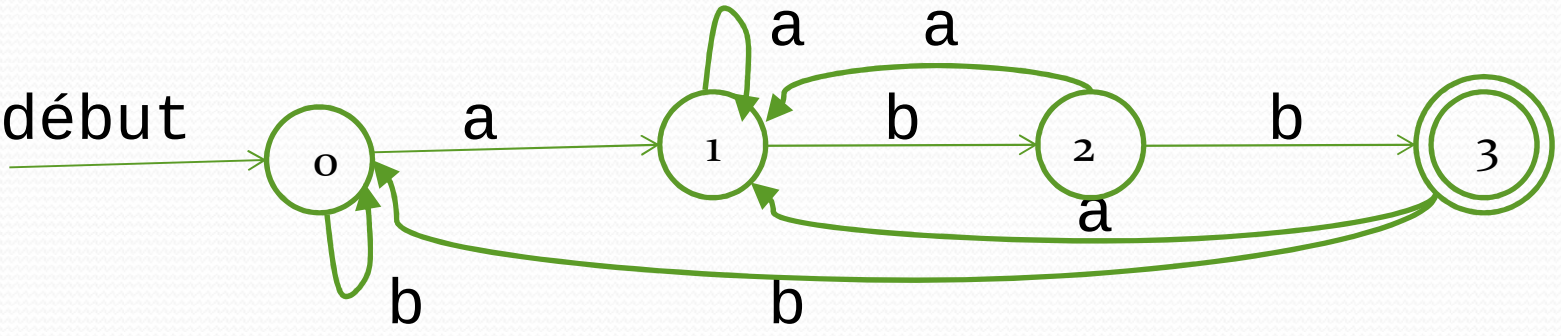
Un AFN est une représentation abstraite d'un algorithme de reconnaissance des chaînes d'un langage.

Un AFD est un algorithme concret de reconnaissance de chaînes.

Remarque: *Toute expression régulière et tout AFN peuvent être convertis en un AFD.*

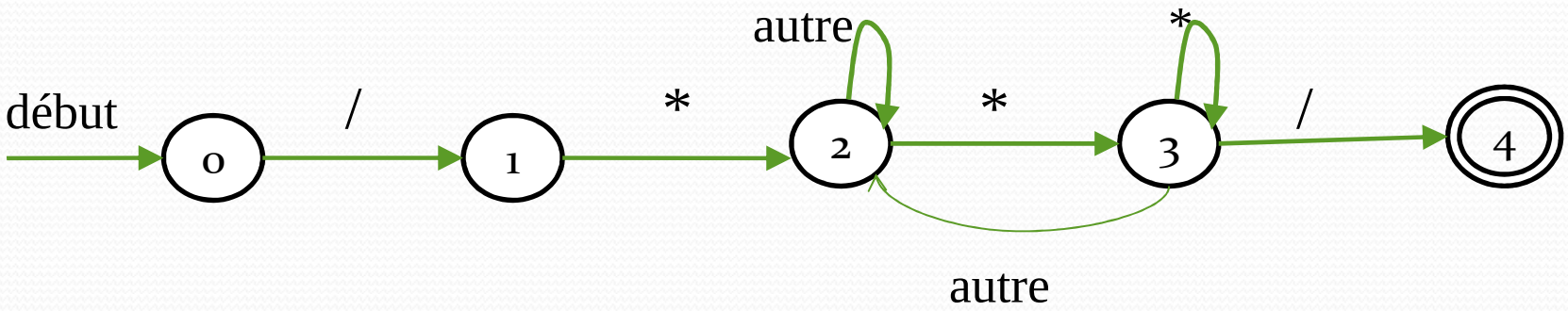
6- Automates à états finis (AEF)

Exemple: L'AFD qui reconnait le langage défini par l'expression régulière : **(a|b)* abb**

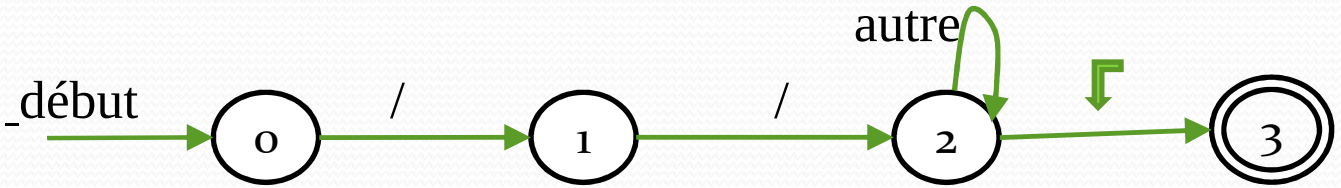


6- Automates à états finis (AEF)

Exemple 2: L'automate à états finis déterministe d'un commentaire à la C



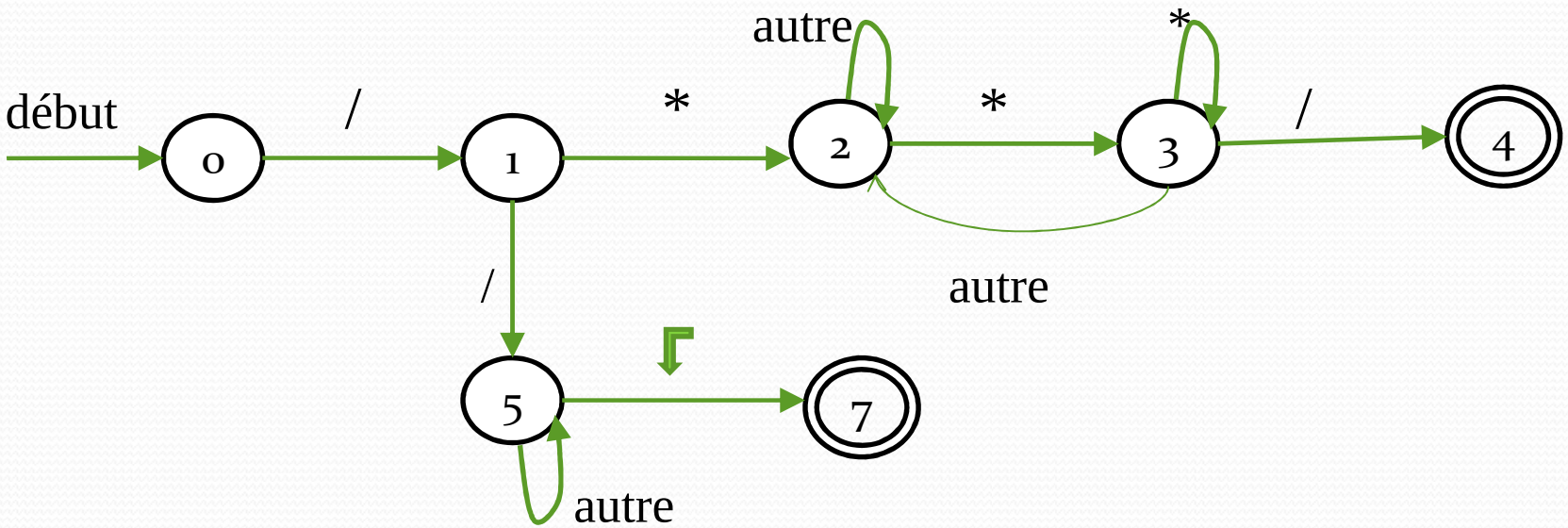
Exemple 3: L'automate à états finis déterministe d'un commentaire à la C++



: désigne le retour à la ligne

6- Automates à états finis (AEF)

Exemple 4: L'automate à états finis déterministe des 2 commentaires groupés.



-

6- Automates à états finis (AEF)

6.4- Algorithme d'application d'un AFD à une chaîne

Donnée:

Une chaîne d'entrée ***x*** terminée par un caractère de fin ***eof***.

Un AFD ***D*** dont l'état initial est e_0, les états finaux sont ***F*** et la fonction de transition est ***trans***.

Résultat:

*"**oui**"* si ***D*** accepte ***x***

*"**non**"* dans le cas contraire.

6- Automates à états finis (AEF)

Méthode:

Appliquer l'algorithme suivant à la chaîne d'entrée ***x***.

La fonction de transition (e,c) donne l'état vers lequel il existe un arc provenant de l'état e pour le caractère c.

La fonction *carcactereSuivant* retourne le caractère suivant de la chaîne d'entrée x.

```
e = e0 ;
c  = caractereSuivant();
tant que (c!=eof){
        s = trans(e,c);
        c=caractereSuivant();
}
si (s ∈ F) retourner "oui"
sinon retourner "non";
```

7- Grammaires régulières

7.1- **Définition**:

Une grammaire est régulière si toutes ses productions vérifient une des 2 formes:

$$A \longrightarrow a\ B$$

ou $$A \longrightarrow a$$

avec **A** et **B** des non-terminaux et **a** un terminal ou ε.

Ces grammaires régulières sont appelées des grammaires *linéaires droites.*

7- Grammaires régulières

Par analogie, il est possible de définir des grammaires *linéaires gauches*:

$$A \longrightarrow B\ a$$

ou $$A \longrightarrow a$$

Remarque:

Les grammaires régulières sont une sous-classe des grammaires hors contextes.

Elles permettent de décrire les langages réguliers.

7- Grammaires régulières

7.2- Correspondance entre une grammaire régulière et un automate:

Nous pouvons faire la correspondance entre un automate et une grammaire régulière de la manière suivante:

- Chaque état de l'automate correspond à un non terminal de la grammaire.
- Chaque transition correspond à une production de la grammaire.
- L'état initial de l'automate correspond à l'axiome de la grammaire.
- Un état final correspond à la production de la chaine vide ε.

7- Grammaires régulières

Exemples:

8- Des expressions régulières aux automates

8.1- Conversion d'un AFN en un AFD:

Il s'agit de remplacer la relation de transition par une fonction partielle qui à un état et un caractère associe au plus un nouvel état.

L'idée est, si l'automate initial est construit sur un ensemble S d'états, de construire un nouvel automate avec comme états des ensembles d'états de S.

8- Des expressions régulières aux automates

8.1- Conversion d'un AFN en un AFD:

Définition:

On définit l'ensemble des ε-successeurs d'un état p et on note ε-Succ(p), l'ensemble des états q tels que:

$$p \xrightarrow{\varepsilon} q$$

On note ε-Succ(P) pour un ensemble d'états, l'union des ε-successeurs des éléments $p \in P$.

Remarque: *De tels successeurs peuvent être obtenus par une plusieurs transitions.*

On définit l'ensemble des successeurs d'un état p pour un caractère a et on note Succ(p,a), l'ensemble des états q tels que:

$$p \xrightarrow{a} q$$

8- Des expressions régulières aux automates

8.1- Conversion d'un AFN en un AFD:

Algorithme: On se donne un automate ($S, e_0, F, Trans$). L'automate correspondant aura pour états des parties de S, c-à-d des ensembles d'états.

On notera de manière générale $P(E)$ l'ensemble des parties E et plus spécifiquement P_S l'ensemble des parties de S.

Un automate déterministe reconnaissant le même langage est:

- ensemble d'états : P_Q
- état initial: $\varepsilon\text{-Succ}(e_0)$
- état d'acceptation: $\{q \subset S \mid q \cap F \neq \emptyset\}$
- transition: $\{(q,a,q') \mid q,q' \in P_Q, a \in A, \forall y \in S.y \in q' \Leftrightarrow$

$$\exists\, x \in q.x \xrightarrow{a} y$$

8- Des expressions régulières aux automates

8.2- Construction d'un AFN à partir d'une expression régulière:

Algorithme de Mc Naughton-Yamada-Thomson:

Données: *Une expression régulière* r *sur un alphabet* Σ.

Résultat: *Un AFN* N *acceptant* $L(r)$.

Méthode:

- *Décomposer* r *en sous expressions constitutives.*
- *Les règles de construction d'un AFN contiennent des règles de base pour traiter les sous-expressions .*

8- Des expressions régulières aux automates

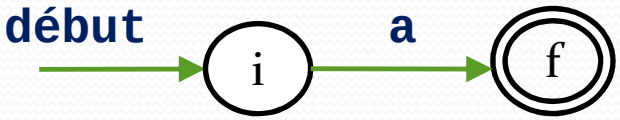
Algorithme de Mc Naughton-Yamada-Thomson:

Soit **r** une expression régulière,

Cas de base:

si **r = ε** , l'automate est:

si **r = a** , l'automate est:

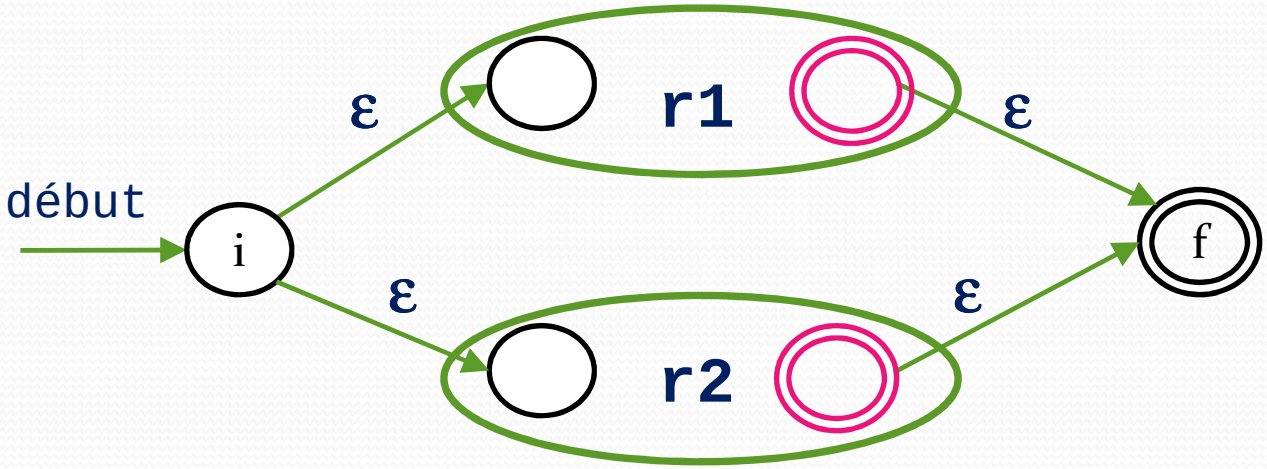


i est l'état initial et f est l'état final de l'AFN.

8- Des expressions régulières aux automates

Cas composés:

1- si **r = r_1 | r_2** , l'automate associé à **r** est :

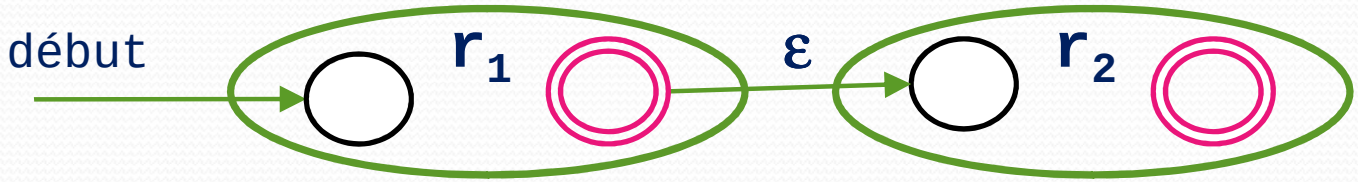


L'état initial associé à **r** comporte des ε-transitions vers les états initiaux des automates associés à r1 et r2.

Les anciens états initiaux deviennent des états ordinaires, de même pour les états finaux

8- Des expressions régulières aux automates

2- si $r = r_1\ r_2$, l'automate associé à r est :

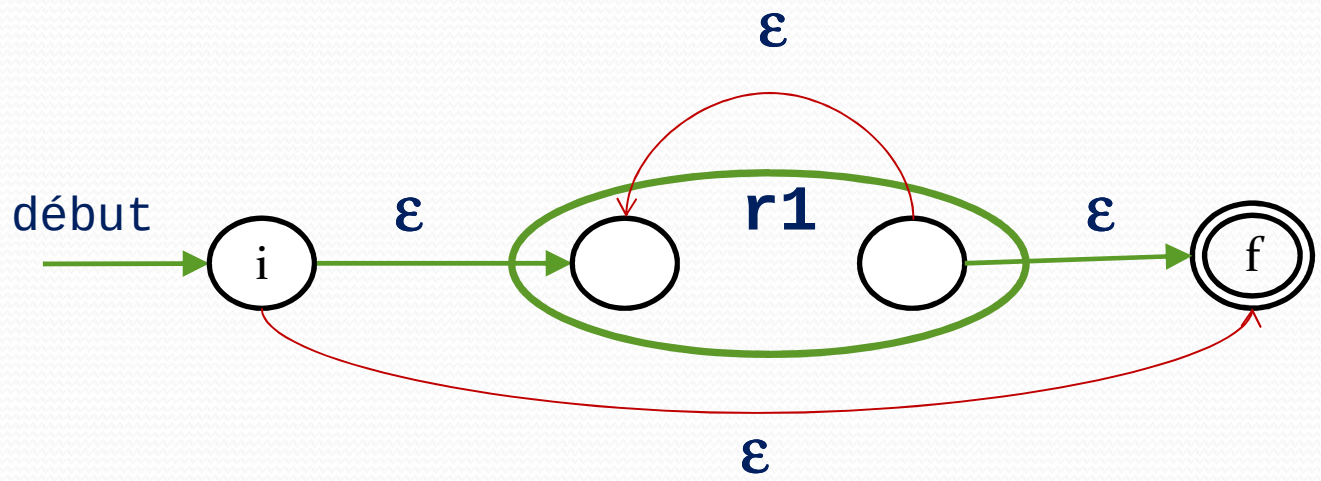


L'état initial associé à r_1 devient un état initial de r et l'état final de r_2 devient état final de r.

.

8- Des expressions régulières aux automates

3- si **r = r_1* = ε | r_1+** , l'automate associé à **r** est :



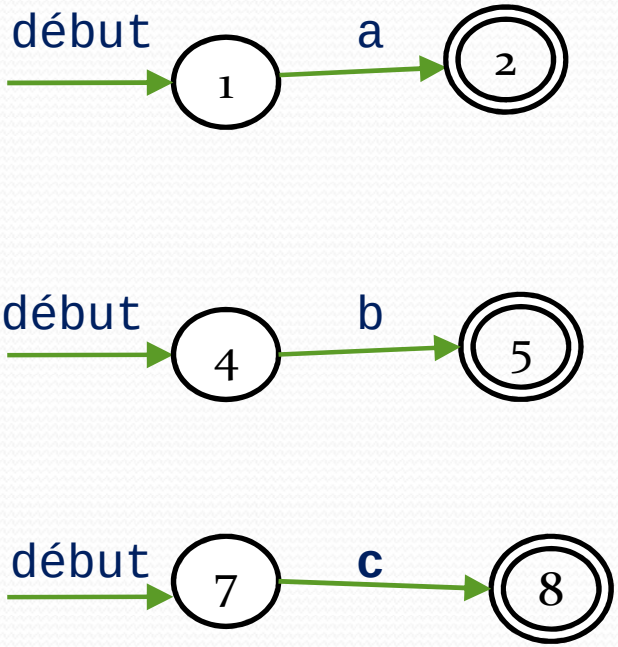
La répétition non nulle (+) consiste à relier l'état final de l'automate de **r1** à son état initial.

Pour ajouter **ε** au langage reconnu par l'automate, il suffit de créer un nouvel état initial et un état final et de les relier avec une transition **ε** .

8- Des expressions régulières aux automates

Exemple: Soit l'expression régulière **`a | b c*`**

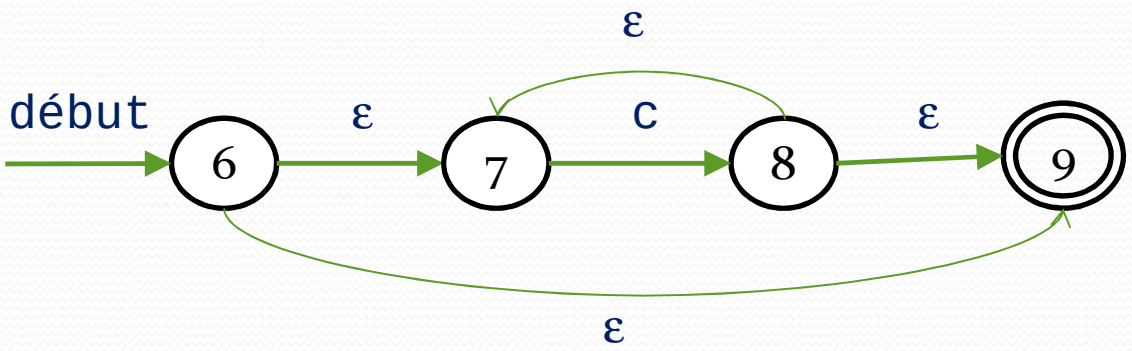
- Pour **`'a'`** , **`'b'`** et **`'c'`**, on a les automates:



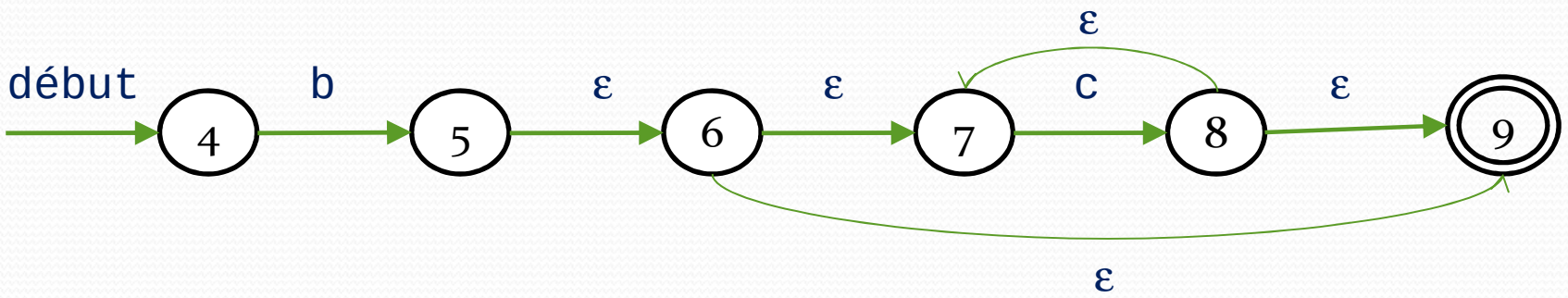
8- Des expressions régulières aux automates

Exemple: Soit l'expression régulière **a | b c***

- Pour **c*** , on a:



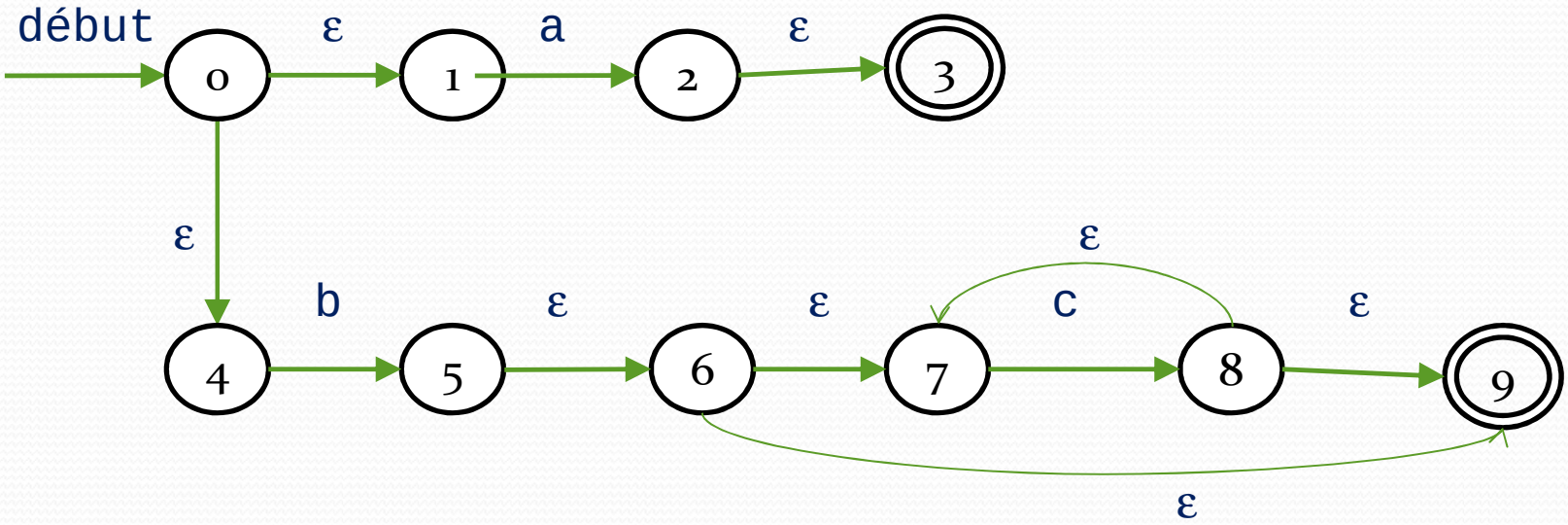
- Pour b c* , on a:



8- Des expressions régulières aux automates

Exemple: Soit l'expression régulière **a | b c***

- Pour **a|bc*** , on a:



L'expression régulière équivalente:

a|b|bcc* = a|b|bc+ = a|bc*

8- Des expressions régulières aux automates

Elimination des ε-transitions:

Elle se fait en 4 étapes:

1- Augmentation des transitions.

2- Propagation des états finaux.

3- Suppression des ε-transitions.

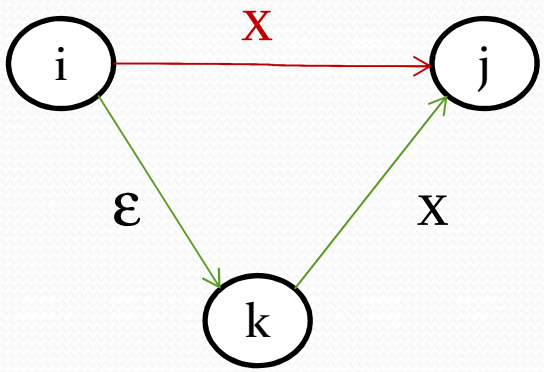
4- Elimination des états inaccessibles.

8- Des expressions régulières aux automates

Elimination des ε-transitions:

1- Augmentation des transitions:

On construit un nouvel automate où il existe une transition entre l'état **i** et l'état **j** étiqueté par **x**, s'il existe un état **k** tel qu'il existe une suite d' ε-transitions de **i** à **k** et qu'il existe une transition **x** de **k** à **j**.

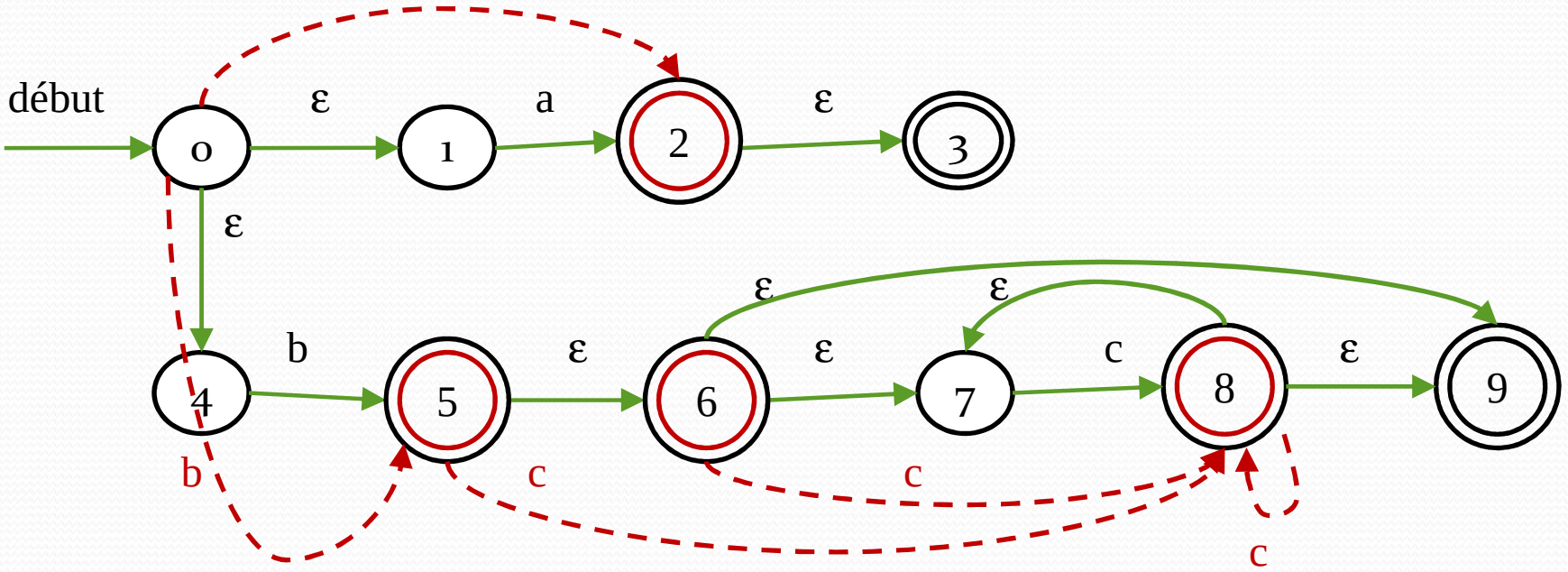


8- Des expressions régulières aux automates

Elimination des ε-transitions:

1- Augmentation des transitions:

2- Un état est final s'il existe est une suite d'ε-transitions qui mènent à un état final.

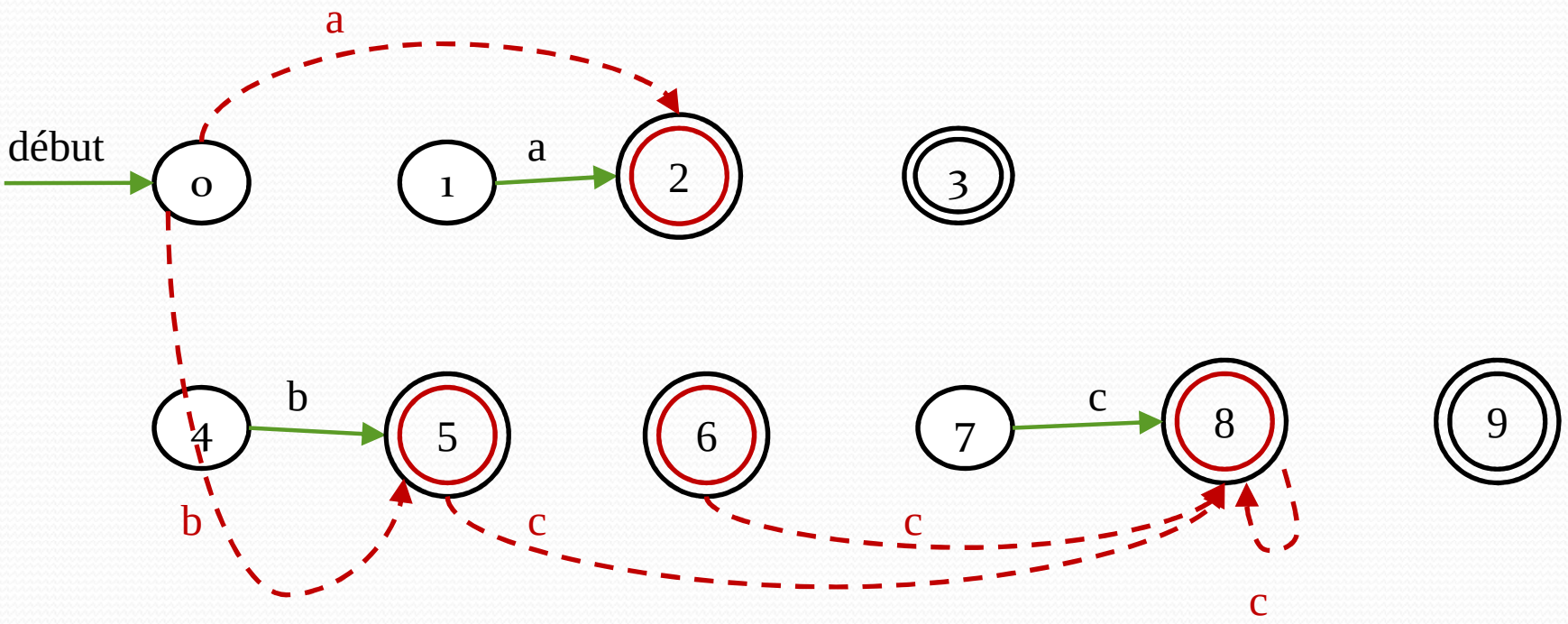


8- Des expressions régulières aux automates

Elimination des ε-transitions:

3- On supprime les ε-transitions:

4- On supprime les états inaccessibles à partir de l'état initial.

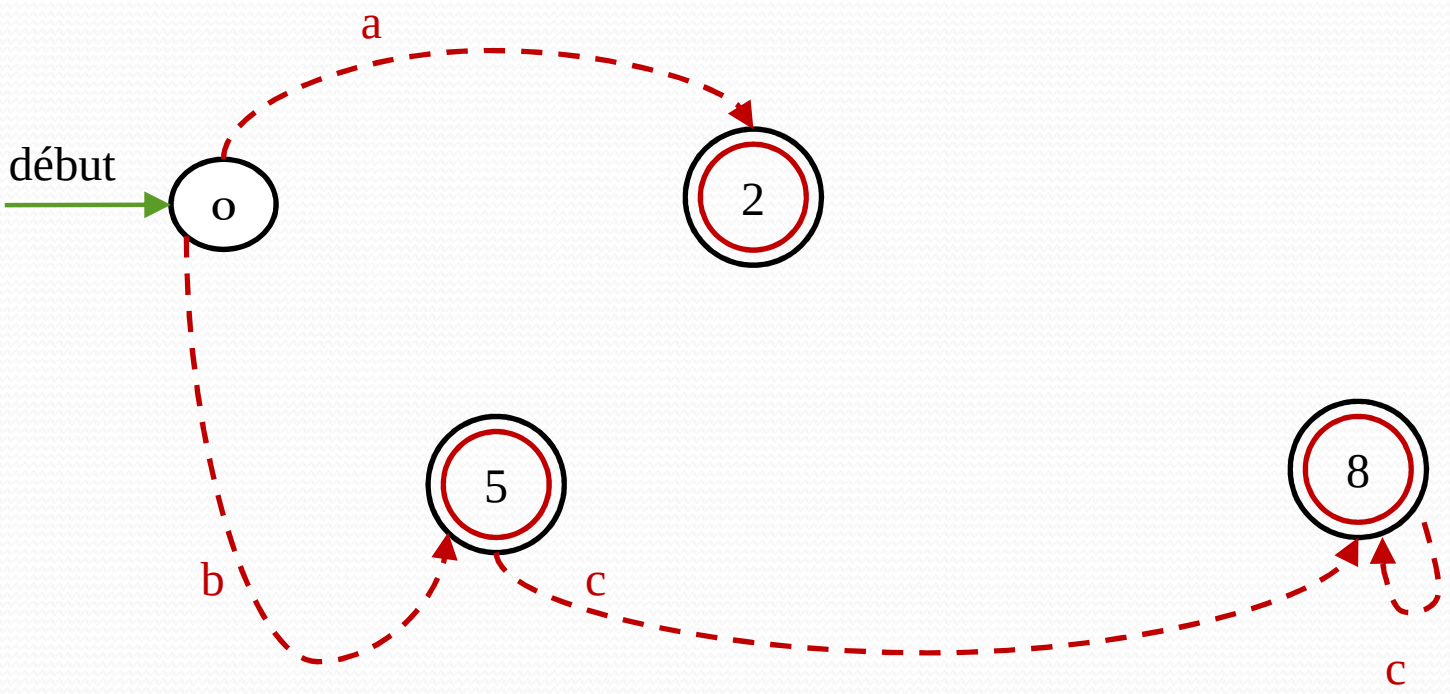


8- Des expressions régulières aux automates

Elimination des ε-transitions:

3- On supprime les ε-transitions:

4- On supprime les états inaccessibles à partir de l'état initial.



8- Des expressions régulières aux automates

8.3- Minimisation du nombre d'états d'un AFD:

But: Obtenir un automate ayant le minimum d'états possible.

Principe: On définit les classes d'équivalence d'états par raffinements successifs.

Chaque classe d'équivalence obtenue forme un seul et même état du nouvel automate.

Méthode:

1- Faire deux classes, A contenant les états finaux et B contenant les états non finaux.

2- S'il existe un symbole....

Le reste est en cours de préparation !

A titre d'exemple, voici le contrôle de l'année dernière

Université Mohammed V-Agdal — Session d'automne 2012
Faculté des sciences — SMI – S5
Département d'Informatique

Contrôle 1 : Compilation
(durée : 1h 30)

Exercice 1 :

Donner les expressions régulières qui décrivent les lexèmes sur l'alphabet {**a**,**b**,**c**} suivants :

1- les lexèmes sur {**a**,**b**,**c**} qui commencent par **b**
2- les lexèmes sur {**a**,**b**,**c**} qui contiennent *exactement* trois **b**
3- les lexèmes sur {**a**,**b**,**c**} qui contiennent *au moins* trois **b**
4- les lexèmes sur {**a**,**b**,**c**} qui contiennent *au plus* trois **b**
5- les lexèmes sur {**a**,**b**,**c**} de longueur deux ne contenant pas la lettre **b**

Exercice 2 :

Simplifier les expressions régulières suivantes :

1- a | aa* | *e*
2- (aa)* | a(aa)*

Exercice 3 :

Soit la grammaire des expressions booléennes suivante :

A ⟶ A ou A | A et A | non A | (A) | vrai | faux

1- Cette grammaire est elle ambigüe ? justifiez votre réponse.
2- Donner une définition dirigée par la syntaxe qui calcule le nombre de « **vrai** » dans une expression.
3- Appliquer cette définition dirigée par la syntaxe pour construire un arbre syntaxique décoré de la phrase: « **non vrai ou faux et vrai** ».