CHAPITRE 2 : ANALYSE LEXICALE

2.1 Rôle de l'analyseur lexical

Il s'agit de transformer des suites de caractères du code source en suite de symboles, correspondant aux unités lexicales, que l'analyseur lexical produit comme résultat de l'analyse. Pour cela, il existe deux approches possibles sachant qu'une passe est définie comme correspondant à un traitement entier d'une représentation du programme source pour produire une représentation équivalente en mémoire secondaire :

- L'analyseur lexical effectue un traitement de la totalité du code source en une première passe, ce qui lui permet d'en obtenir une représentation équivalente sous forme d'une suite d'unités lexicales, sauvegardée dans un fichier en mémoire secondaire. Ce fichier sera l'entrée de l'analyseur syntaxique, qui accomplira son travail pendant une deuxième passe. Le schéma de la figure 2.1 illustre cette approche.
- L'analyseur lexical fonctionne comme une procédure sollicitée à chaque fois, par l'analyseur syntaxique, pour lui fournir une unité lexicale. L'analyseur lexical effectue, pour cela, son travail, pas à pas, en synchronisation avec l'avancement de l'analyse syntaxique. On dit que l'analyse lexicale et syntaxique partagent une même passe. Dans ce cas, il n'est plus question de sauvegarder les unités dans un fichier intermédiaire, cependant, les deux analyseurs (lexical et syntaxique) doivent se trouver ensemble en mémoire. Cette approche est la plus utilisée dans la conception des compilateurs, elle est illustrée par la figure 2.2.

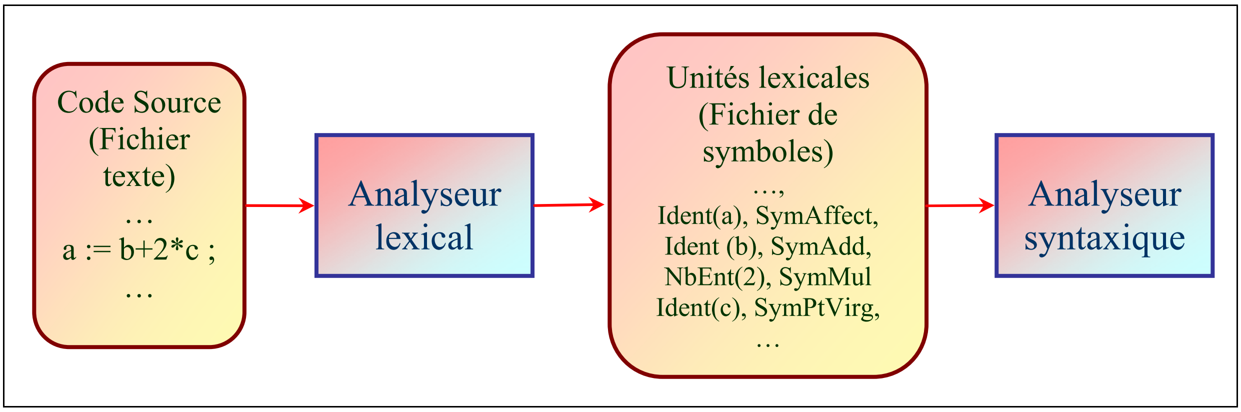


Figure 2.1. Analyse lexicale et syntaxique en deux passes différentes

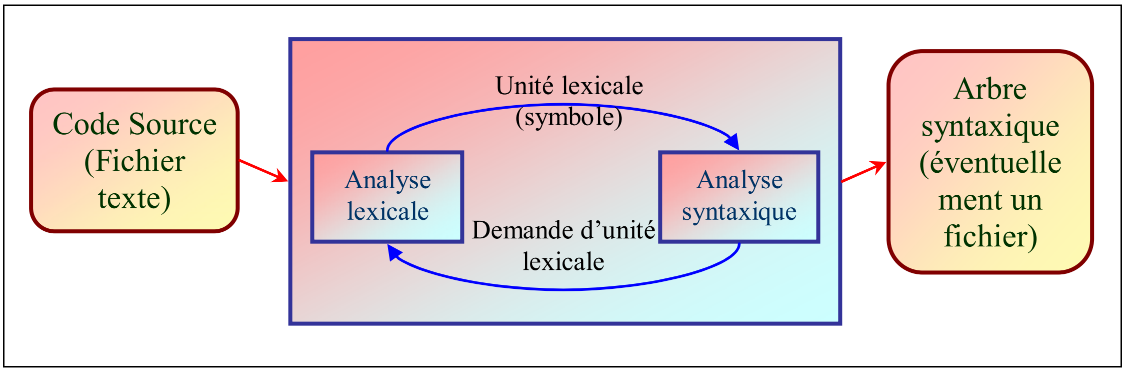


Figure 2.2. Analyses lexicale et syntaxique en une seule passe

2.2 Tâches effectuées par l'analyseur lexical

L'analyse lexicale effectue un ensemble de tâches :

1- Lecture du fichier du code source (fichier texte) caractère par caractère avec éventuellement l'élimination de caractères et informations inutiles (blancs, tabulations, commentaires, caractère de fin de ligne, …) pour réduire la représentation du programme. Une erreur est signalée à ce niveau si un caractère non permis (illégal) par le langage est rencontré.

2- Concaténation des caractères pour former des unités lexicales et signaler, éventuellement, une erreur si la chaîne dépasse une certaine taille.

3- Association d'un symbole à chaque unité lexicale. Cette opération peut engendrer une erreur si l'unité formée ne correspond à aucune unité légale du langage.

4- Enregistrement de chaque unité lexicale et des informations la concernant (nom, valeur, …) dans la table des symboles (en invoquant le gestionnaire de la table des symboles) ou dans un fichier résultat. A ce niveau, il est possible de détecter certaines erreurs telles que la double déclaration d'un identificateur, l'utilisation d'un identificateur sans déclaration préalable, etc.

5- Création d'une liaison entre les unités lexicales d'une ligne et la ligne du fichier la contenant. Cette opération, qui se fait par une simple numérotation des lignes du texte du code source, permet la localisation des lignes en cas d'erreur. Dans certains compilateurs, l'analyseur lexical est chargé de créer une copie du programme source en y intégrant les messages d'erreurs lexicales.

Les symboles associés aux unités lexicales correspondent à la nature de celles-ci. On distingue plusieurs catégories de symboles :

1- Les mots réservés du langage (if, then, begin pour le langage Pascal, par exemple)

2- Les constantes (3, 3.14, True, 'Bonjour', …)

3- Les identificateurs qui peuvent être des noms de variables, de fonctions, de procédures, etc.

4- Les symboles spéciaux, en tenant compte des possibilités d'assemblage de caractères tels que < > <= <> : := + - /*

5- Les séparateurs tels que ; (point virgule) et . (point).

2.3 Spécification des unités lexicales

L'ensemble des unités lexicales, qu'un analyseur reconnaît, constitue un langage régulier L qui peut être décrit par des expressions régulières et reconnu par un automate d'états fini.

La spécification ou description des unités lexicales peut donc être effectuée en utilisant une notation appelée **expressions régulières**. Une expression régulière est construite à partir d'expressions régulières plus simples, en utilisant un ensemble de règles de définition qui spécifient comment le langage est formé.

Soit le vocabulaire $\sum$ = {a, b, c}

1- ε dénote la chaîne vide

2- L'expression régulière a/b dénote l'ensemble des chaînes {a,b}

3- L'expression a^* dénote l'ensemble de toutes les chaînes formées d'un nombre quelconque de a (pouvant être nul), c à d {ε, a, aa, aaa, …}

4- L'expression a^+ dénote l'ensemble de toutes les chaînes formées d'un nombre quelconque, non nul, de a, c à d {a, aa, aaa, …}. Si r est une expression régulière alors $r^* = r^+/\varepsilon$ et $r^+ = rr^*$

5- La notation r? est une abréviation de r/ε et signifie zéro ou une instance de r

6- La notation [abc], où a, b, c sont des symboles de l'alphabet, dénote l'expression régulière a/b/c. Une classe de caractères [a-z] signifie a/b/c/d/ …/z

Exemple 1

Pour chacune des expressions régulières r_i suivantes, on souhaite déterminer le langage dénoté par r_i.

r1 = (a/b)(a/b)
r2 = $(a/b)^*$
r3 = (a*b*)*
r4 = a/a*b

Chaque langage dénoté par r_i est noté L(r_i)

L(r1) = {aa, ab, ba, bb}

L(r2) = {ε, a, aa, aaa, …., b, bb, bbb, …, ab, aab, …, abb, abbb, ba, bba, bbaa, aba, …}. C'est l'ensemble de toutes les chaînes composées d'un nombre quelconque de a et d'un nombre quelconque de b.

L(r3) = L(r2)

L(r4) = {a, b, ab, aab, aaab, …, aaaa…ab} En plus de la chaîne a, cet ensemble contient des chaînes composées d'un nombre quelconque non nul de a suivis par un b.

Exemple 2

On souhaite déterminer la définition régulière de l'ensemble des identificateurs d'un langage sachant qu'ils sont constitués de suite de lettres et de chiffres commençant par une lettre.

Cette définition régulière peut être donnée par :

- lettre = A/B/…/Z/a/b/…/z ou bien [A-Za-z]
- chiffre = 0/1/2/3/4/5/6/7/8/9 ou bien [0-9]
- ident = lettre (lettre/chiffre)* qu'on peut noter directement ident = [A-Za-z] [A-Za-z0-9]*

Remarque

Les expressions régulières sont un outil puissant et pratique pour définir les constituants élémentaires des langages de programmation. Cependant, le pouvoir descriptif des expressions régulières est limité car elles ne peuvent pas être utilisées dans certains cas tels que :

- La définition des constructions équilibrées ou imbriquées. Par exemple, pour les chaînes contenant obligatoirement pour chaque parenthèse ouvrante une parenthèse fermantes, les expressions régulières ne permettent pas d'assurer cela, on dit qu'elles ***ne savent pas compter***.
- La définition d'expressions contenant des répétitions de sous chaînes à différente positions (par exemple des chaînes de la forme $\alpha c \alpha de \alpha$ où α est une combinaison des caractères a et b).
- La définition d'expressions contenant à différentes positions des séquences de caractères ayant la même longueur (par exemple $a^n b^n$ où n désigne le nombre de répétitions du caractère qui le précède).

On peut, cependant, utiliser les expressions régulières dans le cas d'un nombre **fixe** de répétitions d'une construction donnée.

2.4 Approches de construction des analyseurs lexicaux

Après la description des unités lexicales, la construction de l'analyseur lexical peut être effectuée selon l'une des manières suivantes :

1- Une approche simplifiée (manuelle) basée sur les diagrammes de transition.

2- Une approche plus rigoureuse basée sur les automates d'états finis.

3- Une approche utilisant un outil générateur d'analyseurs lexicaux tel que Lex, FLex, JFLex...

Dans les sections suivantes nous décrivons chacune des trois approches.

2.5 Construction d'analyseur lexical basée sur les diagrammes de transition

Les diagrammes de transition sont des organigrammes stylisés (respectant une forme stricte) dont la construction représente une étape préparatoire pour la réalisation d'un analyseur lexical. Le diagramme est ensuite converti en un programme qui permet de reconnaître les unités lexicales exprimées par le diagramme.

2.5.1 Introduction aux diagrammes de transitions

Formellement, un diagramme de transition est un diagramme contenant des cercles numérotés qui représentent des états et des arcs étiquetés matérialisant les transitions entre les états.

Les états du diagramme de transition correspondent aux états de l'analyseur lexical et chaque arc étiqueté par un caractère donné correspond à la transition qu'effectuera l'analyseur suite à la rencontre de ce caractère. Ainsi, au fur et à mesure de la reconnaissance d'une unité lexicale, l'analyseur passe d'un état à un autre.

Considérons deux états e1 et e2 reliés par un arc étiqueté par le caractère c. Etant dans l'état e1 à un moment donné, l'analyseur passera à l'état e2 lorsqu'il lit le caractère c dans le texte source.

Nous adopterons les notations suivantes dans la suite de cette section.

- Un état particulier du diagramme représentera l'état initial où commence la reconnaissance d'une unité lexicale. Il est noté par un état auquel abouti un arc étiqueté par **début**.
- Les états finaux sont identifiés par deux cercles concentriques (cercle doublé) et correspondent, chacun, à la reconnaissance complète d'une unité lexicale.
- Si un ensemble d'arcs sortant d'un état e1 comprend un arc ayant l'étiquette **autre**, elle désigne tous les caractères autres que ceux indiqués explicitement sur les autres arcs sortant de e1.

Remarque

On suppose que les diagrammes de transition utilisés sont **déterministes**. En d'autres termes, aucun caractère ne peut apparaître comme étiquette de deux ou plusieurs arcs quittant un même état.

Exemple 1

Le diagramme de transition de la figure 2.3, correspond à l'expression régulière lettre(lettre/chiffre)* et permet de représenter les identificateurs utilisés dans les langages tel que Pascal (voir exemple 2, section 2.3).

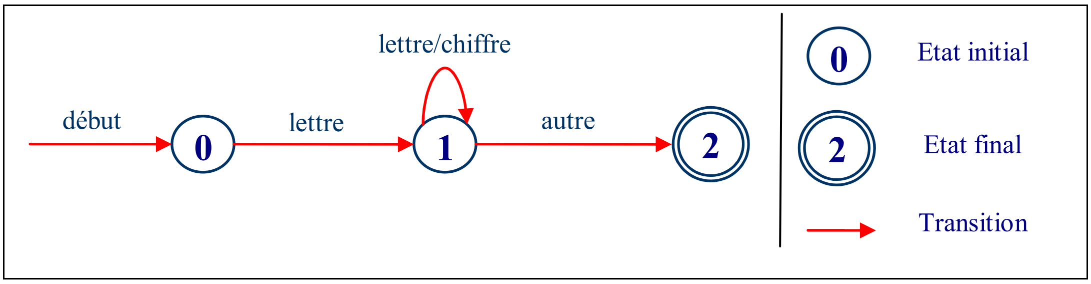


Figure 2.3. Diagramme de transition des identificateurs

Exemple 2

On souhaite donner la définition régulière ainsi que le diagramme de transition qui représente l'ensemble des nombres réels non signés comme 2006, 12.33, 314.3E-2, 0.314E+1, 0.314E2, 314E-2

La définition régulière est donnée par :

- chiffre = 0/1/2/3/4/5/6/7/8/9 ou bien [0-9]
- chiffres = chiffre chiffre* ou bien chiffre+
- fractionOpt = .chiffres/ε ou bien (.chiffres)?
- exposantOpt= (E(+/-/ε) chiffres) / ε ou bien (E(+/-)?chiffres)?
- NbRéel = chiffres fractionOpt exposantOpt

Pour faciliter la représentation par un diagramme de transition, la spécification précédente peut être réécrite directement sous la forme :

NbRéel = chiffre chiffre* (. chiffre chiffre*)? (E(+/-)?chiffre chiffre*)?

Cette forme peut être représentée directement de la figure 2.4.

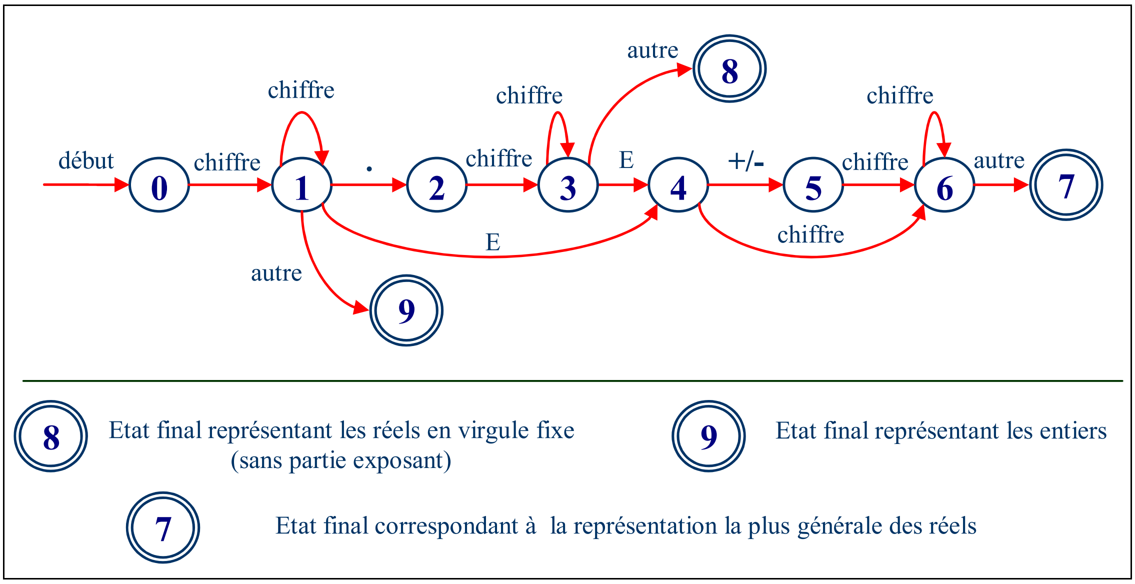


Figure 2.4. Diagramme de transition des nombres réels

2.5.2 Exemple d'implémentation d'un diagramme de transition

L'implémentation du diagramme de transition d'un identificateur (voir figure 2.3) peut être décrite comme suit. Nous utiliserons, pour des raisons de simplification, une série de procédures et fonctions que nous décrivons en premier.

- **lettre** est une fonction qui retourne vrai si le caractère courant passé dans ch est un caractère alphabétique.
- **carsuiv** est une procédure qui retourne le caractère suivant du texte source dans ch (passé en paramètre).
- **retournerId** permet de retourner l'identificateur reconnu. Pour cela, elle associe le symbole correspondant (id) à l'unité formée, range l'identificateur dans la table des symboles, (s'il n'y est pas déjà) et retourne son adresse.
- **chiffre** est une fonction qui retourne vrai si le caractère ch (passé en paramètre) est un chiffre.
- **échec** est une procédure qui permet de passer à un autre diagramme de transition (s'il y en a) ou d'appeler la procédure **erreur**.

Fonction lettre (ch : caractère) : booléen ;

Début

Si (ch ≥ 'A' et ch ≤ 'Z') **ou** (ch ≥ 'a' et ch ≤ 'z') **Alors** lettre ← Vrai

Sinon lettre ← Faux ;

Fin ;

Fonction chiffre (ch : caractère) : booléen ;

Début

Si (ch ≥ '0' Et ch ≤ '9') **Alors** chiffre ← Vrai

Sinon chiffre ← Faux ;

Fin ;

Etapes de l'implémentation

$Etat_0$: carsuiv (ch) ;

Si lettre(ch) alors Aller à $état_1$

Sinon échec() ;

$Etat_1$: carsuiv (ch) ;

Si lettre(ch) Ou chiffre(ch) alors Aller à $état_1$

Sinon Aller à $état_2$;

$Etat_2$: retournerId()

2.5.3 Démarche générale d'implémentation d'un analyseur lexical à partir d'un diagramme de transition

D'une manière plus générale, une suite de diagrammes de transition peut être convertie en un programme qui recherche les unités lexicales selon l'approche suivante :

- Chaque état donne lieu à un segment de programme.
- S'il y a des arcs qui quittent un état, son segment de programme doit inclure la lecture d'un caractère et son test pour sélectionner un arc à suivre.
- S'il y a un arc étiqueté par le caractère lu, ou par une classe de caractères contenant le caractère lu, alors le contrôle est transféré au code correspondant à l'état pointé par cet arc.
- S'il n'existe pas un arc correspondant à la valeur de ch et si l'état actuel n'appartient pas à l'ensemble des états qui indiquent la rencontre d'une unité lexicale (i.e. ce n'est pas un état d'acceptation), alors il y a un échec de l'analyse et la recherche est initialisée à celle de l'unité lexicale spécifiée par le prochain diagramme de transition.
- On utilise généralement une instruction de choix multiple pour trouver l'état de départ du diagramme de transition suivant. On suit les arcs du diagramme de transition en sélectionnant le fragment de code d'un état et on exécute ce fragment pour déterminer le prochain état.

2.6 Construction d'analyseur lexical basée sur les automates finis

2.6.1 Démarche générale de construction d'un analyseur lexical

En se basant sur les propositions démontrables suivantes :

- L'ensemble des unités lexicales d'un langage donné constitue un langage régulier L.
- Pour toute expression régulière r, il existe un automate fini non déterministe (AFN) qui accepte l'ensemble régulier décrit par r.
- Si un langage L est accepté par un automate fini non déterministe (AFN), alors il existe automate fini déterministe (AFD) acceptant L.

On peut définir une approche rigoureuse pour la construction d'un analyseur lexical en utilisant les automates d'états finis. Cette approche est constituée de 6 étapes :

Etape 1 : Spécification des unités lexicales
Spécifier chaque type d'unité lexicale à l'aide d'une expression régulière (ER).

Etape 2 : Conversion ER en AFN
Convertir chaque expression régulière en un automate d'états fini (non déterministe).

Etape 3 : Réunion des AFNs
Construire l'automate Union de tous les automates de l'étape 2
(on peut ajouter un nouvel état initial d'où partent un ensemble d'arcs étiquetés ε).

Etape 4 : Déterminisation ou transformation de l'AFN obtenu en AFD
Rendre l'automate de l'étape 3 déterministe.

Etape 5 : Minimisation de l'AFD.
Minimiser l'automate obtenu à l'étape 4.

Etape 6 : Implémentation de l'AFD minimisé.
Implémenter l'automate obtenu à l'étape 5 en le simulant à partir de sa table de transition.

2.6.2 Automates à états finis

On transforme une expression régulière en un reconnaisseur en construisant un diagramme de transition généralisé appelé automate à états finis. Ce dernier peut être déterministe (AFD) ou non déterministe (AFN).

	AFN	**AFD**
Temps de reconnaissance	Augmente (à cause des retours arrière)	Diminue (un seul chemin possible pour une chaîne donnée)
Espace occupé	Nombre d'états plus petit que celui de l'AFD équivalent	Nombre d'états généralement plus grand que celui de l'AFN équivalent

La transformation d'un automate à états finis en un analyseur lexical consiste à associer une unité lexicale à chaque état final et à faire en sorte que l'acceptation d'une chaîne produise comme résultat l'unité lexicale associée à l'état final en question. Pour cela il faut implémenter la fonction de transition de l'automate en utilisant une table de transition.

2.6.2.1 Automate fini non déterministe

Un AFN est défini par :

- Un ensemble d'état E
- Un ensemble de symboles d'entrée ou alphabet Σ
- Un état initial e_0
- Un ensemble F d'états finaux ou d'acceptation
- Une fonction de transition *Transiter* qui fait correspondre à chaque couple (état,symbole), un ensemble d'états.

Exemple

Considérons l'AFN qui reconnaît le langage décrit par l'expression régulière (a|b)*abb (voir figure 2.5). Pour cet automate :

E = {0, 1, 2, 3} e_0 = 0 Σ = {a, b} F = {3}

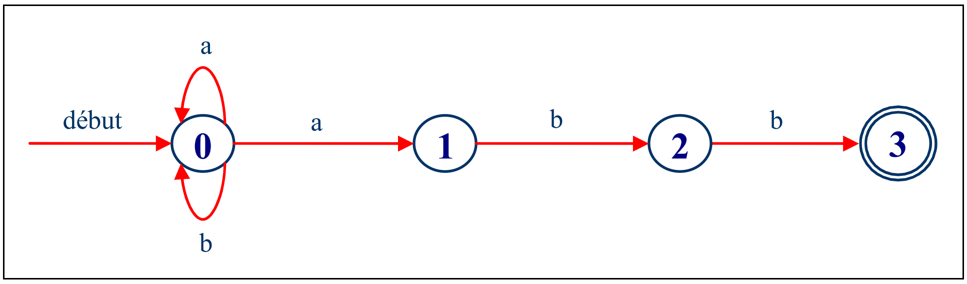


Figure 2.5. Un AFN pour l'expression (a|b)*abb

L'implémentation la plus simple d'un AFN est une table de transition dans laquelle il y a une ligne pour chaque état et une colonne pour chaque symbole d'entrée (avec la chaîne vide ε si nécessaire). L'entrée pour la ligne **i** et le symbole **a** donne l'ensemble des états (ou un pointeur vers cet ensemble) qui peuvent être atteints à partir de l'état **i** avec le symbole **a**. Pour l'AFN de la figure 2.5, la table de transition est la suivante (sachant que la ligne 3 peut être éliminée) :

Etats	Symboles	
	a	b
0	{0, 1}	{0}
1	/	{2}
2	/	{3}
3	/	/

Un AFN accepte une chaîne si et seulement s'il existe au moins un chemin correspondant à cette chaîne entre l'état initial et l'un des états finaux. Un chemin peut être représenté par une suite de transitions appelées *déplacements*. Par exemple, pour la reconnaissance de la chaîne *aabb* par l'AFN de la figure 2.5, il existe deux chemins possibles. Le premier conduit à l'acceptation de la chaîne alors que le second stationne sur l'état 0 qui n'est pas un état final (voir figures 2.6 et 2.7).

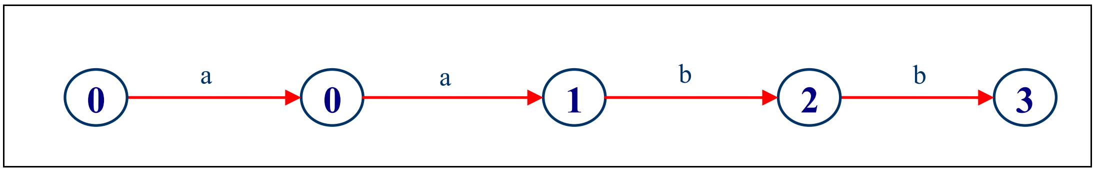


Figure 2.6. Déplacements réalisés en acceptant la chaîne aabb

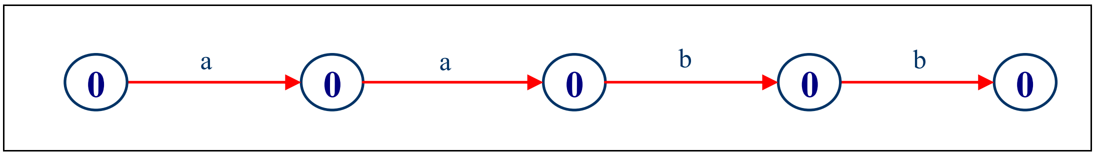


Figure 2.7. Stationnement sur l'état 0 pour la chaîne aabb

2.6.2.2 Automate fini déterministe

Un AFD est un cas particulier d'AFN dans lequel :

- Aucun état n'a de ε-transition
- Pour chaque état **e** et chaque symbole d'entrée **a** il y a au plus un arc étiqueté **a** qui quitte **e**

Dans la table de transition d'un AFD, une entrée contient un état unique au maximum (les symboles d'entrée sont les caractères du texte source), il est donc très facile de déterminer si une chaîne est acceptée par l'automate vu qu'il n'existe, au plus, qu'un seul chemin entre l'état initial et un état final étiqueté par la chaîne en question.

Remarque : La table de transition d'un AFN pour un modèle d'expression régulière peut être considérablement plus petite que celle d'un AFD. Cependant, l'AFD présente l'avantage de pouvoir reconnaître des modèles d'expressions régulières plus rapidement que l'AFN équivalent.

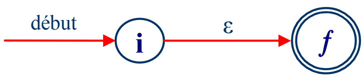
2.6.3 Construction d'AFN à partir d'expressions régulière (étape 2)

Il existe plusieurs algorithmes pour effectuer cette construction. Parmi les plus simples et les plus faciles à implémenter on cite l'algorithme de construction de **Thompson**. Dans la suite, nous utiliserons les notations suivantes :

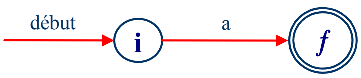
- r : désigne une expression régulière
- *N(r)* : l'AFN correspondant à r (qui reconnaît r)

Les règles de l'algorithme de Thompson sont les suivantes :

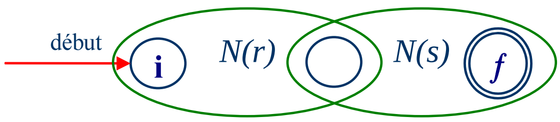
1. Pour r = ε   *N(ε)* est représenté par:



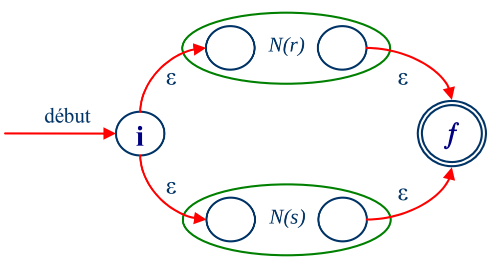
2. Pour r = a   *N(a)* est représenté par:



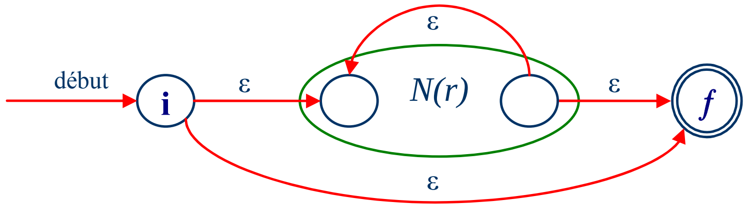
3. Pour *N(r s)* on a :



4. Pour *N(r/s)* on a :



5. Pour r*, N(r*) sera donnée par la représentation suivante :



Exemple

L'AFN qui reconnaît l'expression régulière (a|b)*abb selon la méthode de construction de Thompson est donné dans la figure 2.8 :

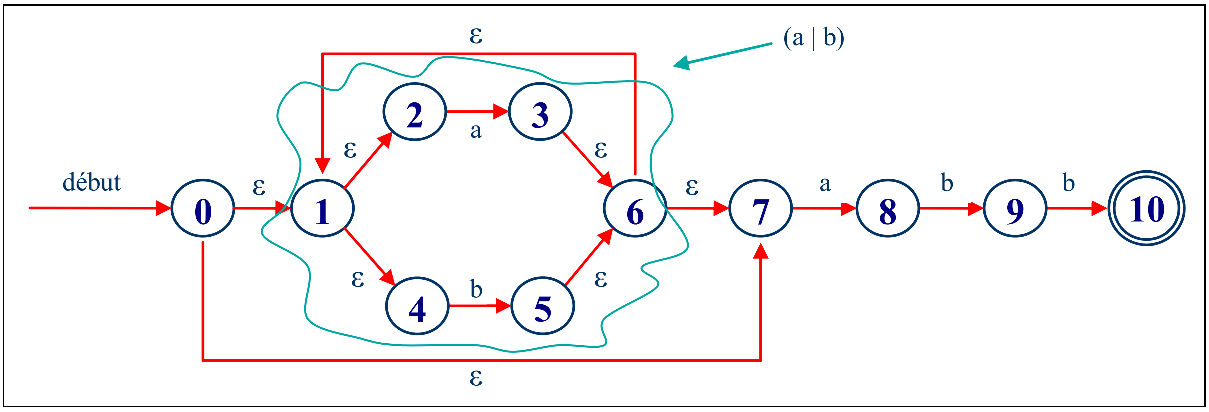


Figure 2.8. Automate fini non déterministe N pour accepter (a|b)*abb

2.6.4 Déterminisation d'un AFN (étape 4)

La déterminisation consiste à transformer un AFN en un AFD équivalent. Elle peut être effectuée par différentes méthodes, nous en présentons trois dans ce qui suit.

2.6.4.1 Déterminisation d'un AFN ne contenant pas de ε-transitions

La déterminisation d'un AFN ne contenant pas de ε-transitions peut être effectuée en appliquant l'algorithme suivant sur la table de transition de l'AFN pour en déduire celle de l'AFD équivalent :

Etapes de l'algorithme

1- Partir de l'état initial $E_0 = \{e_0\}$

2- Construire E_1 qui est l'ensemble des états obtenus à partir de E_0 par la transition étiquetée **a,**

E_1= Transiter(E_0, a).

3- Recommencer l'étape 2 pour toutes les transitions possibles et pour chaque nouvel ensemble d'états E_i.

4- Tous les ensembles d'états E_i contenant au moins un état final deviennent finaux.

5- Renuméroter les ensembles d'états obtenus en tant que simples états.

Exemple
Appliquons l'algorithme précédent sur l'AFN dont e_0 =0 et F = {2, 3} et ayant la table de transition suivante :

Etats	Symboles	
	a	**b**
0	{0, 2}	{1}
1	{3}	{0, 2}
2	{3, 4}	{2}
3	{2}	{1}
4	/	{3}

Après application de l'algorithme, nous obtenons la table de transition de l'AFD suivante :

Etats	Symboles	
	a	**b**
{0}	{0, 2}	{1}
{0, 2}	{0, 2, 3, 4}	{1, 2}
{1}	{3}	{0, 2}
{0, 2, 3, 4}	{0, 2, 3, 4}	{1, 2, 3}
{1, 2}	{3, 4}	{0, 2}
{3}	{2}	{1}
{1, 2, 3}	{2, 3, 4}	{0, 1, 2}
{3, 4}	{2}	{1, 3}
{2}	{3, 4}	{2}
{2, 3, 4}	{2, 3, 4}	{1, 2, 3}
{0, 1, 2}	{0, 2, 3, 4}	{0, 1, 2}
{1, 3}	{2, 3}	{0, 1, 2}
{2, 3}	{2, 3, 4}	{1, 2}

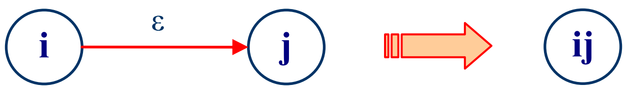
Après renumérotation la table de l'AFD devient :

Etats	Symboles	
	a	**b**
A	B	C
B	D	E
C	F	B
D	D	G
E	H	B
F	I	C
G	J	K
H	I	L
I	H	I
J	J	G
K	D	K
L	M	K
M	J	E

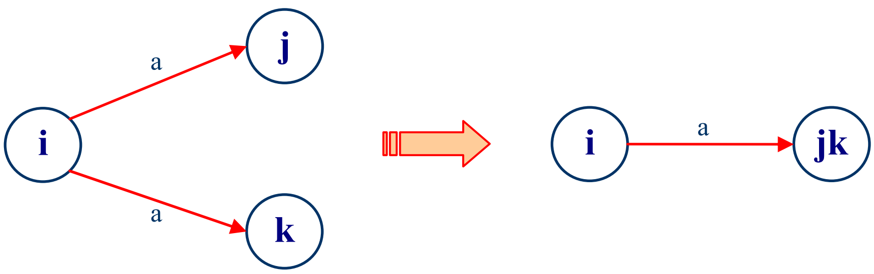
2.6.4.2 Transformation manuelle d'un AFN en AFD (cas général)

On utilise les règles de conversion suivantes pour rendre déterministe un AFN pouvant contenir des ε-transitions:

1. Si on peut passer d'un état i à l'état j avec une transition étiquetée ε alors les états i et j appartiennent à la même classe.



2. Si l'état i mène à l'état j et l'état k par des transitions identiques (même étiquette), alors les états j et k appartiennent à la même classe.



3. Une classe d'état est finale dans l'AFD si elle contient au moins un état final de l'AFN.

Ainsi, l'AFD équivalent à un AFN original possède des états qui sont des classes (regroupements) d'états de l'automate original. Les nœuds de l'AFD correspondent à des sous ensembles de nœuds de l'AFN qui ne sont pas forcement disjoints.

Si l'AFN à n nœuds, l'AFD peut en avoir 2^n. Donc l'AFD peut être beaucoup plus volumineux que l'AFN équivalent, cependant, en pratique le nombre d'états est souvent plus petit que 2^n.

Exemple

Considérons l'automate non déterministe de la figure 2.9, qui reconnaît l'ensemble des chaînes {a, aa, ab, aba, abb}.

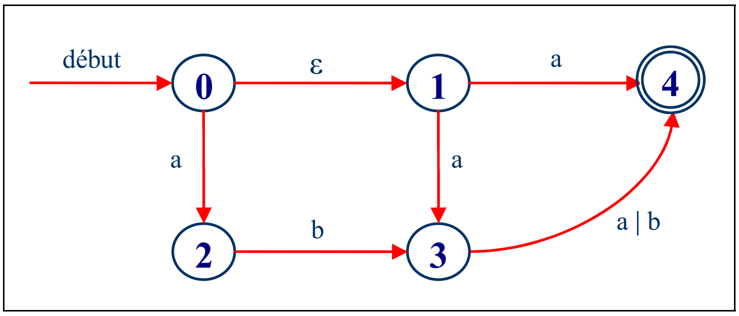


Figure 2.9. Exemple d'AFN

L'AFD correspondant est donné par la figure 2.10, on constate que les états ne sont pas disjoints (234, 34, 4).

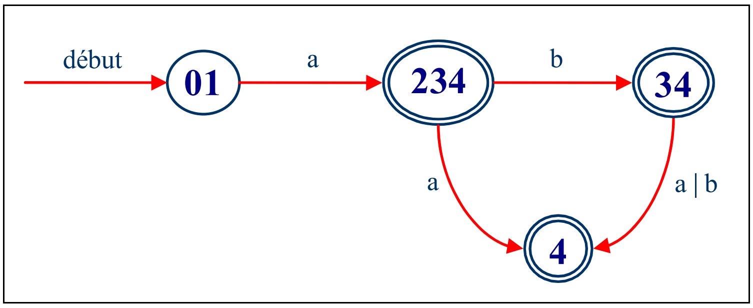


Figure 2.10. AFD équivalent à l'AFN de la figure 2.9

2.6.4.3 Méthode de construction de sous-ensembles pour la transformation des AFN en AFD (cas général)

La construction manuelle décrite précédemment (voir section 2.6.4.2) est simple, mais difficile à appliquer lorsque le nombre d'états est important. Un algorithme plus formel peut être utilisé pour cette étape. Ce dernier est connu sous le nom d'algorithme de *construction de sous-ensembles* et permet de construire une table de transition de l'AFD en se basant sur celle de l'AFN équivalent.

a- Notations et opérations utilisées

Soit un AFN noté N et D son AFD équivalent.

- DTrans est la table des transitions de D
- Détats est l'ensemble des états de D
- e est un état de N, e_0 est l'état initial de N
- T est un ensemble d'états de N
- L'opération ε-fermeture(e) est l'ensemble des états de N accessibles à partir de l'état e par des ε-transitions uniquement.

Remarque

Un état est accessible à partir de lui-même par une ε–transition (même si l'arc n'est pas visible).

- L'opération ε -fermeture(T) donne l''ensemble des états de N accessibles à partir des états e (e $\in$ T) par des ε-transitions uniquement.

Dans l'AFN représenté par la figure 2.11, ε-fermeture(1) = {1, 2, 3, 4, 8}, ε-fermeture(2) = {2, 3, 4}, ε-fermeture({2, 5}) = {2, 3, 4, 5, 6, 7}, ε-fermeture({3, 6}) = {3, 4, 6, 7}

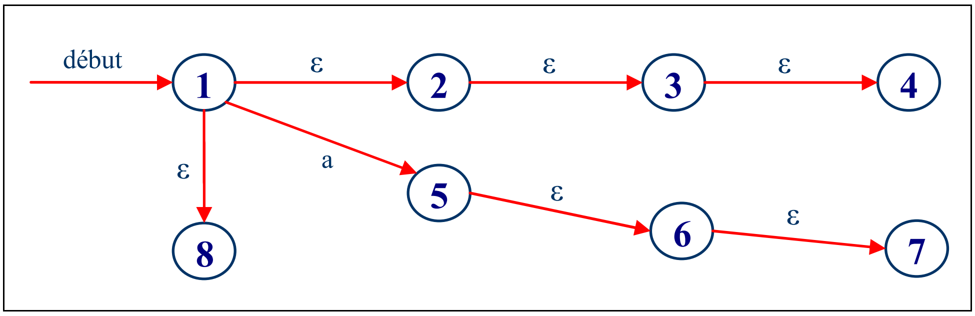


Figure 2.11. Exemple d'AFN pour le calcul ε-fermeture

- L'opération *Transiter(T,a)* donne l'ensemble des états de *N* vers lesquels il existe une transition avec le symbole d'entrée **a** à partir des états $e \in T$.

Dans l'AFN représenté par la figure 2.12, Transiter({1, 5}, a) = {3, 8, 6}, Transiter({1}, b) = {7}, Transiter({2}, b) = { }

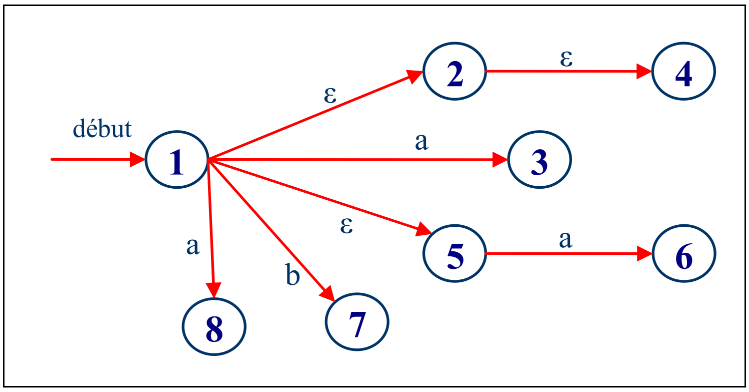


Figure 2.12. Exemple d'AFN pour le calcul de *Transiter*

b- Principe de l'algorithme de construction de sous ensembles

Chaque état de D correspond à un sous ensemble d'états de N de sorte que D simule en parallèle tous les déplacements possibles que N peut effectuer pour une chaîne d'entrée donnée. Ceci signifie que chaque état de D correspond à un ensemble d'états de N dans lesquels N pourrait se trouver après avoir lu une suite donnée de symboles d'entrée, en incluant toutes les ε-transitions possibles avant ou après la lecture des symboles.

L'état de départ de D est ε-fermeture de e_0. Un état de D est final s'il correspond à un ensemble d'états de N contenant, au moins, un état final. On ajoute des états et des transitions à D en utilisant l'algorithme de construction de sous ensembles qui se présente comme suit :

```
Algorithme ConstructionSousEnsembles ;
Début
  ε-fermeture(e0) est l'unique état de Détats et il est non marqué;
  Tantque il existe un état non marqué dans Détats Faire
  Début
      marquer T ;
      Pour chaque symbole d'entrée a faire
      Début
          U ← ε-fermeture(Transiter(T,a)) ;
          Si U∉ Détats Alors Ajouter U comme nœud non marqué à Détats
          DTrans[T, a] ← U ;
      Fin ;
  Fin ;
Fin ;
```

Exemple : Application de l'algorithme de construction de sous ensemble sur l'AFN obtenu dans la figure 2.8.

ε-fermeture(0) = {0, 1, 2, 4, 7} = A

Pour T=A

Marquer A

ε-fermeture(Transiter(A, a)) = ε-fermeture({3, 8}) = {3, 6, 1, 2, 4, 7, 8} = {1, 2, 3, 4, 6, 7, 8} = B

DTran [A, a] ← B

ε-fermeture(Transiter(A, b)) = ε-fermeture({5}) = {5, 6, 1, 2, 4, 7} = {1, 2, 4, 5, 6, 7} = C

DTran [A, b] ← C

On continue à appliquer l'algorithme avec les états actuellement non marqués B et C.

Pour T=B

Marquer B

ε-fermeture(Transiter(B, a)) = ε-fermeture({3, 8}) = B

DTran [B, a] ← B

ε-fermeture(Transiter(B, b)) = ε-fermeture({5, 9}) = {5, 6, 1, 2, 4, 7, 9} = {1, 2, 4, 5, 6, 7, 9} = D

DTran [B, b] ← D

Pour T=C

Marquer C

ε-fermeture(Transiter(C, a)) = ε-fermeture({3, 8}) = B

DTran [C, a] ← B

ε-fermeture(Transiter(C, b)) = ε-fermeture({5}) = {5, 6, 1, 2, 4, 7} = {1, 2, 4, 5, 6, 7} = C

DTran [C, b] ← C

On continue à appliquer l'algorithme avec le seul état actuellement non marqué D.

Pour T=D

Marquer D

ε-fermeture(Transiter(D, a)) = ε-fermeture({3, 8}) = B

DTran [D, a] ← B

ε-fermeture(Transiter(D, b)) = ε-fermeture({5, 10}) = {5, 6, 1, 2, 4, 7, 10} = {1, 2, 4, 5, 6, 7, 10} = E

DTran [D, b] ← E

On continue à appliquer l'algorithme avec le seul état actuellement non marqué E.

Pour T=E

Marquer E

ε-fermeture(Transiter(E, a)) = ε-fermeture({3, 8}) = B

DTran [E, a] ← B

ε-fermeture(Transiter(E, b)) = ε-fermeture({5}) = C

DTran [E, b] ← C

Après plusieurs itérations, on arrive ainsi à un point où tous les sous ensembles d'états de l'AFD sont marqués. A la fin les cinq ensembles d'états construits sont :

- A = {0, 1, 2, 4, 7}
- B = {1, 2, 3, 4, 6, 7, 8}
- C = {1, 2, 4, 5, 6, 7}
- D = {1, 2, 4, 5, 6, 7, 9}
- E = {1, 2, 4, 5, 6, 7, 10}

A est l'état initial de l'automate et E son état final.

La table DTrans de l'AFD obtenu après l'application de l'algorithme de sous-ensembles à l'AFN de la figure 2.8., est donnée par :

Etats	Symboles	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

La figure 2.13 donne le graphe de transition de l'AFD obtenu :

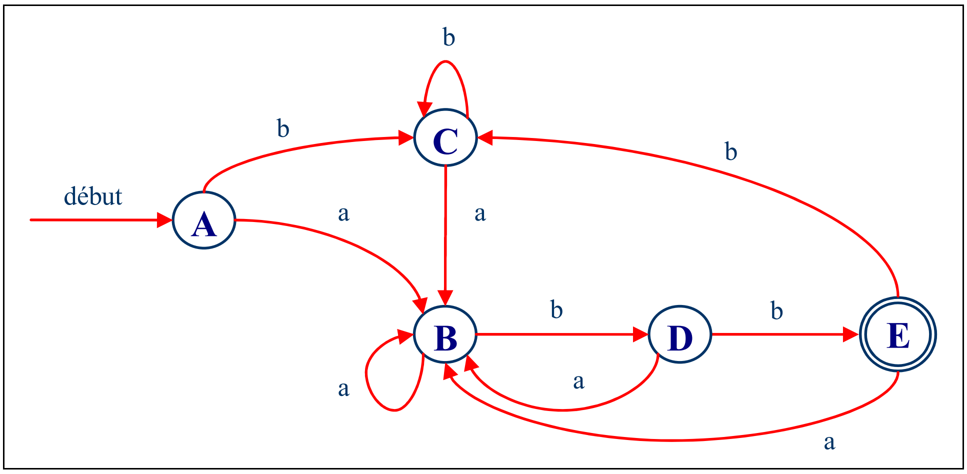


Figure 2.13. AFD équivalent à l'AFN de la figure 2.8

c- Autre formulation de l'algorithme de construction de sous ensembles

L'algorithme de construction de sous-ensembles peut être formulé d'une manière plus condensée en utilisant une représentation tabulaire. Notons que le principe est toujours le même que dans la première formulation présentée précédemment.

1- Commencer avec la ε-fermeture de l'état initial (elle représente le nouvel état initial) ;

2- Rajouter dans la table de transition toutes les ε-fermetures des nouveaux états produits avec leurs transitions ;

3- Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de nouvel état ;

4- Tous les ε-fermetures contenant au moins un état final du premier automate deviennent finaux ;

5- Renuméroter les états en tant qu'états simples.

Exemple : Application de cette formulation de l'algorithme de construction de sous ensemble sur l'AFN obtenu dans la figure 2.8.

La table de transition de l'AFD obtenu est donnée par :

Etats	Symboles	
	a	b
0,1,2,4,7	1,2,3,4,6,7,8	1,2,4,5,6, 7
1,2,3,4,6,7,8	1,2,3,4,6,7,8	1,2,4,5,6,7,9
1,2,4,5,6, 7	1,2,3,4,6,7,8	1,2,4,5,6, 7
1,2,4,5,6,7,9	1,2,3,4,6,7,8	1,2,4,5,6,7,10
1,2,4,5,6,7,10	1,2,3,4,6,7,8	1,2,4,5,6, 7

Après renumérotation des états, nous obtenons la table suivante qui est identique à celle obtenue avec la première formulation de l'algorithme de construction de sous ensembles (voir juste avant la figure 2.13) :

Etats	Symboles	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

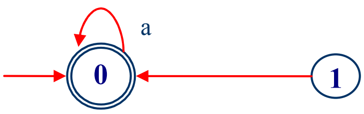
2.6.5 Minimisation d'un AFD (étape 5)

Intuitivement, la minimisation d'un AFD consiste en deux opérations principales :

- Elimination de tous les états inaccessibles (et improductifs) de l'AFD
- Regroupement des états équivalents en les remplaçant par des classes d'équivalence d'états

Sachant que :

- Un état e est accessible s'il existe une chaîne menant à e à partir de l'état initial. Un état, autre que l'état initial, est inaccessible lorsqu'aucun arc n'arrive sur cet état dans un chemin provenant de l'état initial.

  Par exemple, dans l'automate  l'état 1 est inaccessible.

- On dit que deux états sont équivalents si ces états permettent d'atteindre un état final à travers la même chaîne. En d'autres termes, tous les suffixes reconnus à partir de ses états sont exactement les mêmes.

  Par exemple, la figure 2.14 montre la minimisation d'un automate contenant deux états équivalents (états 3 et 4).

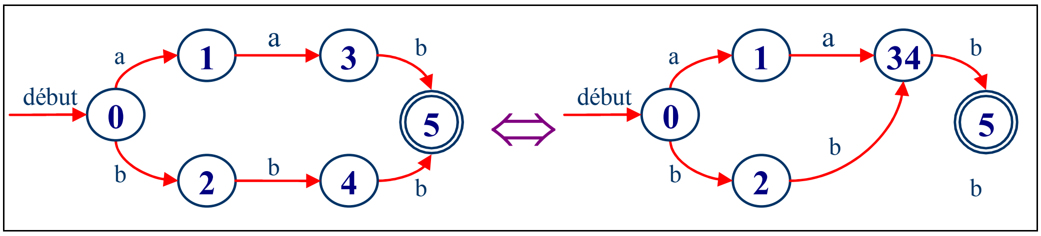


Figure 2.14. Exemple de minimisation d'un AFD

Un algorithme simplifié de minimisation peut être utilisé, il fonctionne par raffinements successifs en définissant des classes d'équivalence d'états qui vont correspondre aux états du nouvel automate (minimisé). Cet algorithme comprend les étapes suivantes :

1. Définir deux classes, C1 contenant les états finaux et C2 les états non finaux.
2. S'il existe un symbole a et deux états e_1 et e_2 d'une même classe tels que Transiter(e_1, a) et Transiter(e_2, a) n'appartiennent pas à la même classe (d'arrivée), alors créer une nouvelle classe et séparer e_1 et e_2. On laisse dans la même classe tous les états qui donnent un état d'arrivée dans la même classe.
3. Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de classes à séparer.
4. Chaque classe restante forme un état du nouvel automate.

2.6.6 Simulation d'un AFD (étape 6)

Soit une chaîne x représentée par un fichier de caractères se terminant par EoF qui est un caractère spécial indiquant la fin d'un fichier. Soit un AFD D avec un état initial e_0 et un ensemble d'état finaux F. Soit la fonction Transiter(e, ch) qui donne l'état vers lequel il y a transition à partir de l'état e avec le caractère ch.

L'algorithme de simulation de l'AFD est le suivant :

```
Algorithme SimulAFD ;
Début
  e ← e0 ;
  carsuiv(ch) ;
  Tantque ch ≠ EoF Faire
  Début
      e ← Transiter(e, ch) ;
      carsuiv(ch) ;
  Fin ;
  Si e ∈ F Alors accepter(x)
  Sinon rejeter(x) ;
Fin ;
```

2.7 Générateurs d'analyseurs lexicaux

Les étapes de construction d'un analyseur lexical à partir de la spécification des unités lexicales peuvent être automatisées en utilisant un outil logiciel dit générateur d'analyseur lexical tel que l'outil LEX qui peut être considéré comme le plus ancien. Cette approche de construction des analyseurs lexicaux est considérée comme la plus simple car on se limite à fournir les définitions exactes des unités lexicales du langage considéré et le générateur fait le reste en générant le code source de l'analyseur lexical.

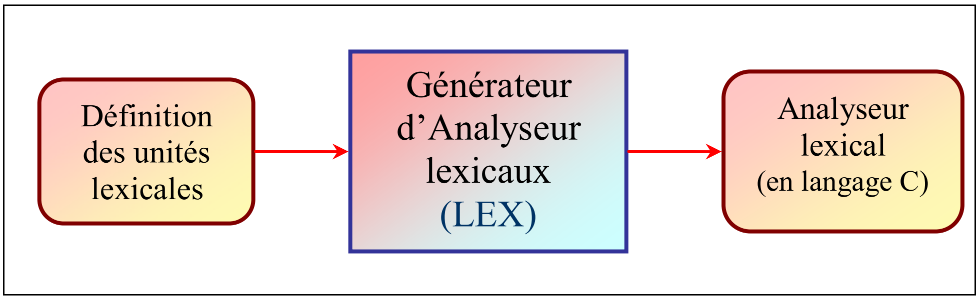


Figure 2.15. Construction d'un analyseur lexical en utilisant un générateur

Le générateur LEX permet de générer un analyseur lexical en langage C à partir des spécifications des unités lexicales exprimées en langage LEX. Le noyau de LEX permet de produire l'automate d'état fini correspondant à la reconnaissance de l'ensemble des unités lexicales spécifiées. Ainsi, LEX construit une table de transition pour un automate à états finis à partir des modèles d'expressions régulières de la spécification en langage LEX.

L'analyseur lexical lui même consiste en un simulateur d'automate à états finis (en langage C), qui utilise la table de transition obtenue pour rechercher les unités lexicales dans le texte en entrée. La spécification des unités lexicales est effectuée en utilisant le langage LEX, en créant un programme *lex.l* constitué de trois parties :

Déclarations
%%
Règles de traduction
%%
Procédures auxiliaires

La première partie contient des déclarations de variables, de constantes et des définitions régulières utilisées comme composantes des expressions régulières qui apparaissent dans les règles de traduction.

Les règles de traduction sont des instructions (en langage C) de la forme m_i {action$_i$} où chaque m_i est une expression régulière et chaque action$_i$ est un fragment de programme qui décrit quelle action l'analyseur lexical devrait réaliser quand une unité lexicale concorde avec le modèle m_i.

Les procédures auxiliaires sont toutes les procédures qui pourraient être utiles dans les actions.

SERIE DE TD N°:1 COMPILATION
ANALYSE LEXICALE

<u>Exercice 1</u>

Donner la définition régulière et le diagramme de transition correspondant à des identificateurs qui ne dépassent pas 4 caractères (lettres ou chiffres) dont le premier est obligatoirement une lettre.
Proposer une implémentation pour le diagramme de transition obtenu (algorithme correspondant à l'analyseur de ce type d'unités lexicales).

<u>Exercice 2</u>

Soit un automate d'états finis dont l'ensemble des états est {0, 1, 2, 3, 4}, état initial: 0, états finaux: 0 et 3, le vocabulaire étant {a, b}. Notons T la fonction de transition de cet automate.

T(0, a) = {2}, T(0, b) = {1}, T(1, a) = {2}, T(1, b) = {1}, T(2, a) = {3}, T(2, b) = {2},

T(3, a) = {2, 4}, T(3, b) = {1, 3}, T(4, a) = {4}, T(4, b) = {4}

1. Etablir la table de transition et la représentation graphique de l'automate.
2. Etablir la table de transition et la représentation graphique de l'automate déterministe équivalent.
3. Donner le chemin de reconnaissance des chaînes *bababb* et *abb* par l'automate déterministe obtenu.
4. Donner le chemin de reconnaissance de la chaîne *aaab* par l'automate initial puis proposer une minimisation de cet automate qui n'affecte pas le langage qu'il reconnaît.

<u>Exercice 3</u>

Effectuer la minimisation des automates suivants :
1). L'AFD de la figure 2.13 (page 23 du support de cours).
2). L'AFD obtenu dans la table en page 17 du support de cours (états de A à M).

<u>Exercice 4</u>

Soient les expressions régulières suivantes: a, abb, a^*b^+
1). Appliquer les étapes 2 à 5 de la démarche de construction d'un analyseur lexical basée sur les automates d'états finis, afin de construire un analyseur lexical pour les trois expressions régulières précédentes, correspondant à des unités lexicales.
2). Décrire l'analyse lexicale des chaînes aaba et abb en donnant leur chemin de reconnaissance (utiliser l'algorithme de simulation d'un AFD qui représente la $6^{ème}$ étape de construction de l'analyseur).

<u>Exercice 5</u>

Soit un langage, représentant un petit sous-ensemble du langage Pascal, et constitué par les unités lexicales suivantes :

- Des identificateurs commençant par une lettre suivie d'une combinaison quelconque de lettres ou de chiffres
- Des constantes numériques entières non signées (sans limitation de longueur)
- L'affectation (:=)

Construire l'analyseur lexical correspondant à ce langage en utilisant l'approche basée sur les automates d'états finis.