Livret – 14

Programmer avec des grammaires

automate à pile de mémoire, automate d'états finis, construction d'un interpréteur

Outil utilisé : C#

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

SOMMAIRE

14.1 : Programmation avec des C-grammaires

Plan du chapitre:

1. Programmation par les grammaires (programme en C#)

D'un point de vue pratique, les grammaires sont un outil abstrait puissant. Nous avons vu qu'elles permettaient de décrire des langages de quatre catégories. Elles servent aussi :

- soit à générer des phrases dans le langage engendré par la grammaire (en ce sens elles permettent la programmation),
- soit à analyser un énoncé quelconque pour savoir s'il appartient ou non au langage engendré par la grammaire (en ce sens elles permettent la construction des analyseurs et des compilateurs).

Nous adoptons ici le point de vue " mode génération " d'une grammaire afin de s'en servir comme d'un outil de spécification sur les mots du langage engendré par cette grammaire. On appelle aussi cette démarche *programmation par la syntaxe*.

Nous nous restreindrons au C-grammaires et aux grammaires d'états finis.

Soit $G = (V_N, V_T, S, R)$, une telle grammaire et L(G) le langage engendré par G.

Objectif : Nous voulons construire un programme qui nous exhibe sur l'écran des mots du langage L(G).

1.1 Méthode pratique de programmation avec un langage récursif

$G = (V_N, V_T, S, R)$ à traduction en programme pratique en langage X générateur de mots.

La grammaire G est supposée ne pas contenir de règle récursive gauche (du genre $A \rightarrow A\alpha$), sinon il faut essayer de la changer ou abandonner.

1° Tous les éléments du vocabulaire auxiliaire V_N deviennent les noms d'un bloc-procédure du programme.

2° Le vocabulaire terminal V_T est décrit soit par un type prédéfini du langage X s'il est simple, sinon par une structure de donnée et un TAD.

3° Toutes les règles de G sont traduites de cette manière :

3.1° le symbole de V_N de la partie Gauche de la règle indique le nom du bloc-procédure que l'on va implanter.
3.2° la partie droite d'une règle correspond à l'implémentation du corps du bloc-procédure, pour chaque symbole α de cette partie droite si c'est :

- un élément de V_T, il est traduit par une écriture immédiate de sa valeur (généralement un **écrire**(α) traduit dans le langage X).
- un élément de V_N, il est traduit par un appel au bloc-procédure du même nom que lui.

4° Le bloc-procédure représentant l'axiome S est appelé dans le programme principal. Chaque appel de S fournira un mot du langage L(G).

Afin de bien persuader le lecteur de la non dépendance de la méthode vis à vis du langage nous construisons l'exemple en Delphi et en C#.

Exemple fictif :

grammaire	*Traduction en Delphi*	*Traduction en C#*
G = (V_N,V_T,S, R) V_N = { S, A, B } V_T = { a, b } **Axiome** : S **Règles** : ..... k : S → aAbBb	V_N → **procedure** S ; V_T → **Type** Vt = char ; **procedure** A ; **procedure** B ;	V_N → **void** S( ) ; V_T → **char** ; **void** A( ) ; **void** B( ) ;

La règle k est traduite par l'implantation du corps du bloc-procédure associé à l'axiome S (partie gauche):

règle	*Traduction en Delphi*	*Traduction en C#*

k : S → aAbBb	**procedure** S ;
	begin
	writeln('a') ;
	A ;
	writeln('b') ;
	B ;
	writeln('b') ;
	end;
		void S ( ) {
		
		System.out.println('a');
		A( ) ;
		System.out.println('b');
		B( ) ;
		
		}

Le lecteur comprend ici le pourquoi de la contrainte de règles non récursives gauches (du genre A → A α), le bloc-procédure s'écrirait alors :

règle	*Traduction en Delphi*	*Traduction en C#*

A → A α	**procedure** A ;
	begin
	A ;
	...
	end ;
		void A ( )
		{
		A( ) ;
		...
		}

Ce qui conduirait le programme à un empilement récursif infini du bloc-procédure A (limité par la saturation de la pile d'exécution de la machine avec production d'une exception de débordement de pile).

1.2 Application de la méthode : un mini-français

Etant donné G une grammaire d'un sous-ensemble du français dénommé **mini-fr**.

G = (V_N,V_T,S,R)
V_T ={**le, un, chat, chien, aime, poursuit, malicieusement,**
joyeusement, gentil, noir, blanc, beau, ' . '}
V_N = { ⟨*phrase*⟩, ⟨*GN*⟩, ⟨*GV*⟩, ⟨*Art*⟩, ⟨*Nom*⟩, ⟨*Adj*⟩, ⟨*Adv*⟩, ⟨*verbe*⟩ }
Axiome : ⟨ *phrase* ⟩

Règles :

1 : ⟨ *phrase* ⟩→⟨ *GN* ⟩⟨ *GV* ⟩⟨ *GN* ⟩ .
2 : ⟨ *GN* ⟩→⟨ Art ⟩⟨ *Adj* ⟩⟨ *Nom* ⟩
3 : ⟨ *GN* ⟩→⟨ *Art* ⟩⟨ *Nom* ⟩⟨ *Adj* ⟩
4 : ⟨ *GV* ⟩→⟨ *verbe* ⟩ | ⟨ *verbe* ⟩⟨ *Adv* ⟩
5 : ⟨ *Art* ⟩→ **le** | **un**
6 : ⟨ *Nom* ⟩→ **chien** | **chat**
7 : ⟨ *verbe* ⟩→ **aime** | **poursuit**
8 : ⟨ *Adj* ⟩→ **blanc** | **noir** | **gentil** | **beau**
9 : ⟨ *Adv* ⟩→ **malicieusement** | **joyeusement**

Traduisons à l'aide de la méthode précédente, cette grammaire G en un programme C# générant des phrases de L(G).

A) les procédures du programme

Chaque élément de $\mathbf{V_N}$ est associé à une procédure :

$\mathbf{V_N}$ = {⟨phrase⟩, ⟨GN⟩, ⟨GV⟩, ⟨Art⟩, ⟨Nom⟩, ⟨Adj⟩, ⟨Adv⟩, ⟨verbe⟩}

$\mathbf{V_N}$	*Traduction en Delphi*	*Traduction en C#*
$\mathbf{V_N}$ = {⟨phrase⟩, ⟨GN⟩, ⟨GV⟩, ⟨Art⟩, ⟨Nom⟩, ⟨Adj⟩, ⟨Adv⟩, ⟨verbe⟩}	**procedure** phrase; **procedure** GN; **procedure** GV; **procedure** Art; **procedure** Nom; **procedure** Adj; **procedure** Adv; **procedure** verbe;	**void** phrase() **void** GN() **void** GV() **void** Art() **void** Nom() **void** Adj() **void** Adv() **void** verbe()

B) les types de données associés à V_T

Nous utilisons la structure de tableau de chaînes, commode à cause de sa capacité d'accès direct pour stocker les éléments de $\mathbf{V_T}$. Toutefois, au lieu de ne prendre qu'un seul tableau de chaînes pour $\mathbf{V_T}$ tout entier, nous partitionnons $\mathbf{V_T}$ en 5 sous-ensembles disjoints :

$\mathbf{V_T}$ = tnom ∪ tadjectif ∪ tarticle ∪ tverbe ∪ tadverbe
où :
tnom={ chat, chien } tadjectif={ blanc, noir, gentil, beau }
tarticle ={ le, un } tverbe ={ aime, poursuit }
tadverbe={ malicieusement, joyeusement }

Spécification d'implantation en C# :

Ces cinq ensembles sont donc représentés en C# chacun par un tableau de chaînes.

```
Const int     Maxnbmot=4; // nombre maximal de mots dans un tableau
```

```
champs
    private   String[ ] tnom ;
    private   String[ ] tadjectif ;
    private   String[ ] tarticle ;
    private   String[ ] tverbe ;
    private   String[ ] tadverbe ;
```

Nous construisons une classe que nous nommons Gener_fr qui est chargée de construire et d'afficher une phrase du langage **mini-fr** :

Tous les champs seront privés la seule méthode publique est la méthode phrase qui traduit l'axiome de la grammaire et qui lance le processus de génération et bienentendu le constructeur d'objet de la classe qui est obligatoirement publique.

Etat de la classe C# à ce niveau de construction :

```
class  Gener_fr {
  const int Maxnbmot=4; // nombre maximal de mots dans un tableau
  private   String[ ] tnom ;
  private   String[ ] tadjectif ;
  private   String[ ] tarticle ;
  private   String[ ] tverbe ;
  private   String[ ] tadverbe ;
  private void GN  ( ) {  }
  private void GV   ( ) {  }
  private void Art  ( ) {  }
  private void Nom  ( ) {  }
  private void Adj  ( ) {  }
  private void Adv  ( ) {  }
  private void verbe  ( ) {  }
  public void phrase   ( ) {  } // axiome de la grammaire
 }
```

C) Initialisation des données associées à V_T

Un ensemble de méthodes de chargement est élaboré afin d'initialiser les contenus des différents tableaux, ce qui permet de changer aisément leur contenu, voici dans la classe Gener_fr les méthodes C# associées :

Initialisation des contenus en C# :

```
void initnom ( )
{
  tnom[0] ="chat";
  tnom[1] = "chien";
}
```

```
void initverbe ( )
{
   tverbe[0] = "poursuit";
   tverbe[1] = "aime";
}
```

```
void initadjectif ( )
{
  tadjectif[0] = "beau";
  tadjectif[1] = "gentil";
  tadjectif[2] = "noir";
  tadjectif[3] = "blanc";
}
```

```
void initarticle ( )
{
  tarticle[0] = "le";
  tarticle[1] = "un";
}
```

```
void initadverbe ( )
{
 tadverbe[0] = "malicieusement";
 tadverbe[1] = "joyeusement";
}
```

```
void initabl ( ){
   initnom ( );
   initarticle ( );
   initverbe ( );
   initadjectif ( );
   initadverbe ( );
}
```

Ces cinq méthodes « initnom », « initverbe », « initadjectif », « initarticle », « initadjectif », sont appelées dans la méthode générale d'initialisation du vocabulaire $\mathbf{V_T}$ tout entier « initabl ».

Nous avons besoin d'une fonction de tirage aléatoire lorsqu'il se présente un choix à faire entre plusieurs règles dérivant du même élément de $\mathbf{V_N}$, comme dans la règle suivante :

règle 4 : ⟨ ***GV*** ⟩ → ⟨ ***verbe*** ⟩ **|** ⟨ ***verbe*** ⟩ ⟨ ***Adv*** ⟩

où nous trouvons deux cas de dérivation possible pour le groupe verbal **GV** :

soit < ***verbe*** >,
soit < ***verbe*** > < ***Adv*** >

Le programme devra procéder à un choix aléatoire entre l'une ou l'autre des dérivations possibles.

Nous construisons une méthode Alea qui reçoit en entrée un entier indiquant le nombre **n** de choix possibles et qui renvoie une valeur aléatoire comprise entre **1** et **n**.

Une implantation possible:

C#

```
private Random ObjAlea = new Random();

int Alea (int n) {
    return ObjAlea.Next(n);
}
```

D) Traduction de chacune des règles de G

Nous traduisons en employant la méthode proposée règle par règle.

REGLE

```
1 : ⟨ phrase ⟩→⟨ GN ⟩⟨ GV ⟩⟨ GN ⟩.
```

Nous construisons le corps de la méthode phrase qui est la partie gauche de la règle. Les instructions correspondent aux appels des procédures GN, GV.

C#

```
void phrase ( )
{
   GN ( );
   GV ( );
   GN ( );
   System.Console.WriteLine('.') ;
}
```

REGLE

```
2 : ⟨ GN ⟩→⟨ Art ⟩⟨ Adj ⟩⟨ Nom ⟩
3 : ⟨ GN ⟩→⟨ Art ⟩⟨ Nom ⟩⟨ Adj ⟩
```

Nous traitons ensemble ces deux règles car elles correspondent à la même procédure de génération du groupe nominal **GN**.

Ici nous avons un tirage aléatoire à faire pour choisir laquelle des deux dérivations le programme utilisera.

C#

```
void GN ( )
{
   if (Alea(2) ==1) // pour  règle 3
   {
       Art ( );
       Nom ( );
       Adj ( );
   }
   else // pour règle 2
   {
       Art ( );
       Adj ( );
       Nom ( );
    }
}
```

REGLE

```
4 : ⟨ GV ⟩→⟨ verbe ⟩ | ⟨ verbe ⟩⟨ Adv ⟩
```

Dans ce cas nous avons aussi à faire procéder à un tirage aléatoire afin de choisir soit :

la dérivation ⟨ GV ⟩→⟨ verbe ⟩
ou bien la dérivation ⟨ GV ⟩→⟨ verbe ⟩⟨ Adv ⟩

C#

```
void GV ( )
{
   if (Alea(2) ==1) // règle: < verbe >
       Verbe ( );
   else // règl:e < verbe >< Adv >.
   {
      Verbe ( );
      Adv ( );
    }
}
```

Les règles suivantes étant toutes des règles terminales, elles sont donc traitées comme le propose la méthode : chaque règle terminale est traduite par un System.Console.WriteLine(a). Lorsqu'il y a plusieurs choix possibles, là aussi nous procédons à un tirage aléatoire afin d'emprunter l'une des dérivations potentielles.

REGLE

```
5 : ⟨ Art ⟩→ le | un
6 : ⟨ Nom ⟩→ chien | chat
7 : ⟨ verbe ⟩→ aime | poursuit
8 : ⟨ Adj ⟩→ blanc | noir | gentil | beau
9 : ⟨ Adv ⟩→ malicieusement | joyeusement
```

C#

```
void Art ( )
{
    System.Console.Write (tarticle[Alea(2)]+' '  ) ;
}
```

```
void Adj ( )
{
    System.Console.Write (tadjectif [Alea(4)]+' '  ) ;
}
```

```
void Verbe ( )
{
    System.Console.Write (tverbe [Alea(2)]+' '  ) ;
}
```

```
void Nom ( )
{
    System.Console.Write (tnom [Alea(2)]+' '  ) ;
}
```

```
void Adv ( )
{
   System.Console.Write (tadverbe [Alea(2)]+' '  ) ;
}
```

Le programme principal se bornera à appeler la procédure **phrase** (l'axiome de la grammaire) à chaque fois que nous voulons engendrer une phrase. Ci-dessous dans le tableau de gauche nous listons la classe Gener_fr comportant toutes les méthodes précédentes et le programme d'instanciation d'un objet de cette classe permettant la génération aléatoire de phrases. A l'identique dans le tableau de droite, nous listons la classe Gener_fr Java, puis une autre classe principale générant une suite de phrases aléatoires :

Classe C#

```
class  Gener_fr
{
 const int Maxnbmot = 4; // nombre maximal de mots dans un tableau
 public Gener_fr()
 {
  initabl();
 }
 private String[] tnom = new String [Maxnbmot];
 private String[] tadjectif = new String [Maxnbmot];
 private String[] tarticle = new String [Maxnbmot];
 private String [] tverbe = new String [Maxnbmot];
 private String [] tadverbe = new String [Maxnbmot];

 private Random ObjAlea = new Random();
```

```
private int Alea(int n)
 {
  return ObjAlea.Next(n);
 }

 private   void initnom( )
 {
  tnom[0] = "chat";
  tnom[1] = "chien";
 }
 private   void initverbe( )
 {
  tverbe[0] = "poursuit";
  tverbe[1] = "aime";
 }
 private void initadjectif()
 {
  tadjectif[0] = "beau";
  tadjectif[1] = "gentil";
  tadjectif[2] = "noir";
  tadjectif[3] = "blanc";
 }
 private void initarticle( )
 {
  tarticle[0] = "le";
  tarticle[1] = "un";
 }
 private void initadverbe( )
 {
  tadverbe[0] = "malicieusement";
  tadverbe[1] = "joyeusement";
 }
 private void initabl( )
 {
  initnom();
  initarticle();
  initverbe();
  initadjectif();
  initadverbe();
 }

 private void GN( )
 {
  if (Alea(2) == 1) // pour  règle 3
  {
    Art();
    Nom();
    Adj();
  }
  else // pour règle 2
  {
    Art();
    Adj();
    Nom();
  }
```

```
 }

private void GV( )
 {
  if (Alea(2) == 1) // règle: < verbe >
    Verbe();
  else // règl:e < verbe >< Adv >.
  {
    Verbe();
    Adv();
  }
 }

 private void Art ( )
 {
  System.Console.Write(tarticle[Alea(2)] + ' ');
 }

 private void Nom ( )
 {
  System.Console.Write(tnom[Alea(2)] + ' ');
 }

 private void Adj ( )
 {
  System.Console.Write(tadjectif[Alea(4)] + ' ');
 }

 private void Adv ( )
 {
  System.Console.Write(tadverbe[Alea(2)] + ' ');
 }

 private void Verbe ( )
 {
  System.Console.Write(tverbe[Alea(2)] + ' ');
 }

 private void fin ( )
 {
  System.Console.Write('.');
 }

 public void phrase() // axiome de la grammaire
 {
  GN();
  GV();
  GN();
  fin();
 }
}
```

Utiliser la classe C#

```
public class UtiliseGenerFr
{
```

```
public static void Main( )
{
  Gener_fr miniFr = new Gener_fr( );
  for(int i=0; i<10; i++)
    miniFr.phrase( );
 }
}
```

```
un chien gentil aime joyeusement le beau chien .
le chien noir aime un chat gentil .
un blanc chien aime le chien gentil .
un beau chat poursuit joyeusement un chien noir .
le chat blanc aime le gentil chat .
un chien beau aime malicieusement un chien gentil .
un noir chat poursuit le beau chien .
un chien blanc aime joyeusement un chien blanc .
le noir chat aime un blanc chien .
un chat noir aime le gentil chat .
un gentil chien aime un blanc chien .
un chat noir aime joyeusement un chien blanc .
le chien blanc aime un blanc chat .
le noir chat poursuit un gentil chat .
un chien blanc aime malicieusement un noir chien .
le blanc chien aime le noir chat .
le gentil chat poursuit le chien gentil .
le chien blanc poursuit joyeusement un chat blanc .
le beau chat aime un chien beau .
un chat blanc aime joyeusement le chien gentil .
```

2. C-grammaires et automates à pile de mémoire

Une C-grammaire est une grammaire de type 2 dans la classification de Chomsky.

Nous adoptons maintenant l'autre point de vue, " **mode analyse** " d'une grammaire, pour s'en servir comme d'un outil de spécification sur la **reconnaissance** des mots du langage engendré par cette grammaire. Cette partie est appelée l'analyse syntaxique.

Dans le cas d'une grammaire de type 3, ce sont les automates d'états finis qui résolvent le problème. Comme ils sont faciles à faire construire par un débutant, nous les avons détaillés dans un paragraphe qui leur est consacré spécifiquement. Dans le cas où G est de type 2 sans être de type 3, nous allons esquisser la solution du problème en utilisant les automates à pile sans fournir de méthodes générales sur leur construction systématique. L'écriture des analyseurs à pile fait partie d'un cours sur la compilation qu'il n'est donc pas question de développer auprès du débutant. Il est toutefois possible de montrer dans le cadre d'une solide initiation sur des exemples bien choisis et simples que l'on peut programmer de tels analyseurs.

Nous retiendrons le côté formateur du principe général de la reconnaissance des mots d'un langage par un ordinateur, aussi bien par les automates d'états finis que par les automates à pile. Nous trouverons aussi une application pratique et intéressante de tels automates dans le filtrage des données. Enfin, lorsque nous élaborerons des interfaces, la reconnaissance de

dialogues simples avec l'utilisateur sera une aide à la convivialité de nos logiciels.

2.1 Définition d'un automate à pile

Un automate à pile est caractérisé par la donnée de six éléments :

A = (V_T, E, q0, F, μ , V_p,)
où :
E= ensemble des états (card E est fini)
q0 ∈ E (**q0** , est appelé l'état *initial*).
E ⊂ **F** (F, est appelé l'ensemble des états *finaux*).
$\mathbf{V_T}$ =Vocabulaire terminal, contient l'élément **#**.
$\mathbf{V_p}$ = Vocabulaire de la pile, contient toujours 2 éléments spéciaux (notés ω et **Nil**).

ω ∈ $\mathbf{V_p}$ (*symbole initial de pile*) **et** **Nil** ∈ $\mathbf{V_p}$

μ : V_T* x E x V_p → E x **Vp' *** (**μ** est appelé *fonction de transition de A*)
avec : **Vp'*** = (**Vp** ∪ { # })* (monoïde sur **Vp** ∪ {#})

Une transition (ou encore règle de transition) a donc la forme suivante :
$\mathbf{\mu : (a_i , q_i , \alpha) \rightarrow \mu(a_i , q_i , \alpha) = (q_k , x_n)}$

Nous trouvons dans ces automates, une pile (du type **pile LIFO**) dans laquelle l'automate va ranger des symboles pendant son analyse.

2.2 Algorithme de fonctionnement d'un automate à pile

En pratique, afin de simplifier les programmes à écrire, nous définirons et utilisons par la suite un vocabulaire de pile $\mathbf{V_p}$ normalisé ainsi :

$\mathbf{V_p = V_p' = V_T \cup \{\#\} \cup \{ \omega , Nil \}}$

Intérêt de la notion d'automate :

C'est la fonction de transition qui est l'élément central d'un automate, elle doit être définie de manière à permettre d'**analyser** un mot de $\mathbf{V_T}$*, et aussi de **décider de l'appartenance** ou non d'un mot à un certain langage. Ce langage d'appartenance est appelé le langage reconnu par l'automate.

Nous construisons notre automate à pile comme étant un dispositif physique muni :

- d'une **bande d'entrée** (de papier, ou magnétique par exemple) composée de cases ne pouvant contenir chacune qu'un seul symbole de $\mathbf{V_T}$ à la fois,

- d'une autre **bande de pile** composée de cases ne pouvant contenir chacune qu'un seul symbole de $\mathbf{V_p}$ à la fois,

- de deux têtes de lecture ou écriture de symboles :

  - l'une de **lecture** capable de reconnaître des éléments du vocabulaire terminal $\mathbf{V_T}$ dans chaque case de la **bande d'entrée**, et **possédant plusieurs états**. La tête de lecture se déplace de gauche à droite d'une case à la fois,

  - l'autre tête de **lecture/écriture** capable de lire ou d'écrire des éléments du vocabulaire de pile $\mathbf{V_p}$ dans chaque case de la **bande de pile**, cette tête se déplace dans les deux sens, mais d'une seule case à la fois.

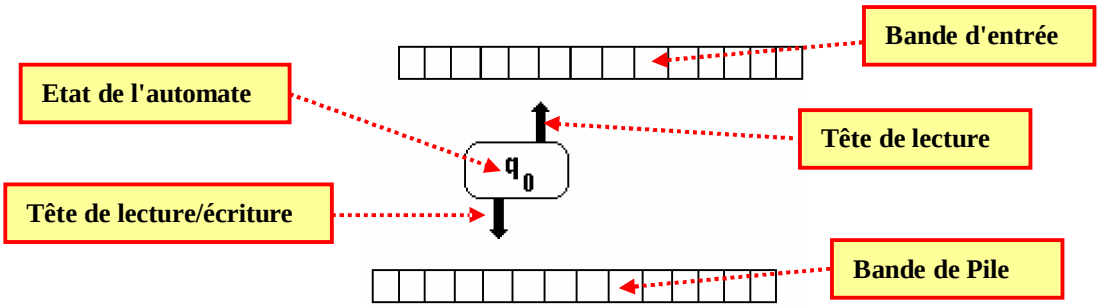
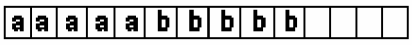


fig - un automate à pile

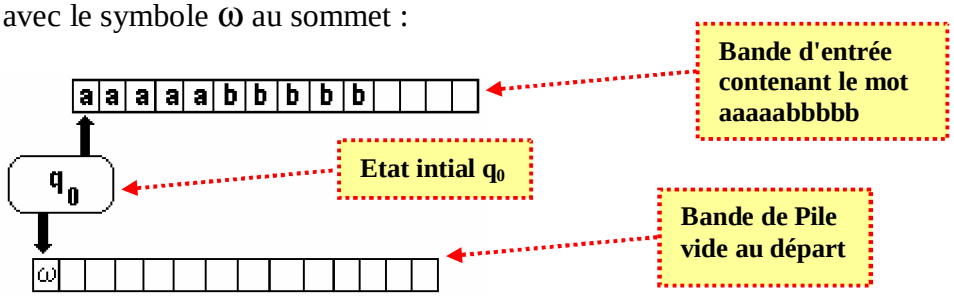
- Les règles de transitions spécifient la manipulation de la bande d'entrée et de la pile de l'automate.

L'algorithme de fonctionnement d'un tel automate est le suivant :

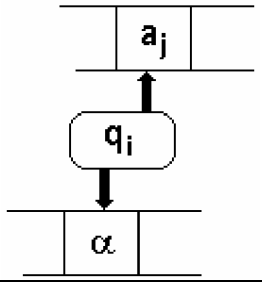
On fournit un mot que l'on écrit symbole par symbole de gauche à droite dans chaque case de l'automate (par exemple avec $\mathbf{V_T}$ ={a,b} le mot **aaaaabbbbb**):



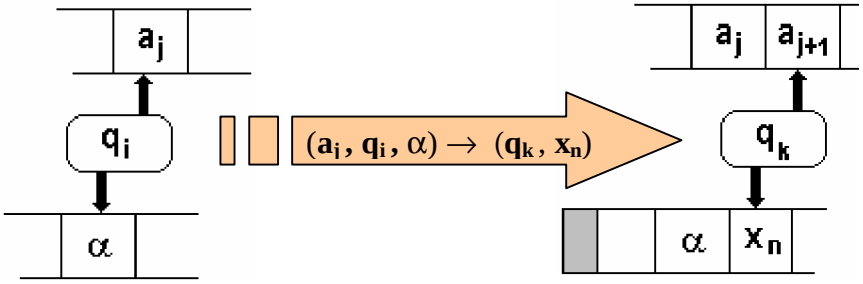
L'automate est mis à l'état initial $\mathbf{q_0}$, sa tête de lecture d'entrée est positionnée sur la première case à gauche de la bande d'entrée(1er symbole du mot à reconnaître), la pile est initialisée avec le symbole ω au sommet :



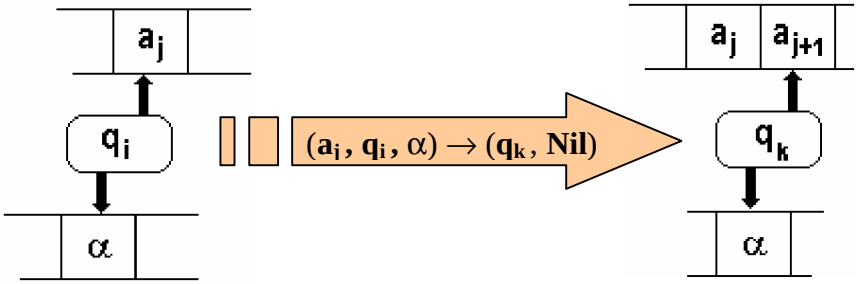
- La tête de lecture se déplace par examen des règles de transition de l'automate en y rajoutant l'examen du sommet de la pile. Le triplet $(\mathbf{a_j, q_i, \alpha})$ enclenche le processus de recherche d'une transition possible dans la partie gauche de la liste des règles de transitions de μ (il y a recherche de la transition μ : $(\mathbf{a_j , q_i , \alpha}) \rightarrow$ .....).

Supposons que la case contienne le symbole $\mathbf{a_j}$ que la tête soit à l'état $\mathbf{q_i}$, et que le sommet de pile ait pour valeur α.	

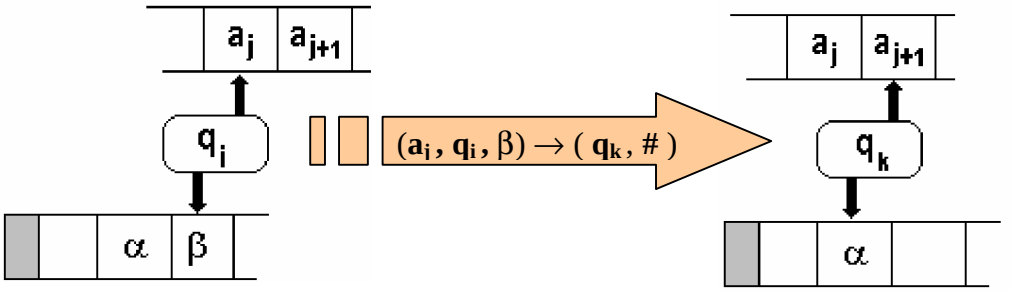
- **La transition $(a_j, q_i, \alpha) \rightarrow (q_k, x_n)$** signifie que l'automate peut passer de l'état $\mathbf{q_i}$ à l'état $\mathbf{q_k}$ à condition que le mot d'entrée débute par la chaîne préfixe $\mathbf{a_j}$ élément de $\mathbf{V_T^*}$ (notons que la chaîne $\mathbf{a_j}$ peut être réduite par sa définition à un seul élément de $\mathbf{V_T}$, ce qui est généralement le cas pratique) et que la chaîne α de **Vp** se trouve en sommet de pile. Le résultat de la transition fait que le symbole $\mathbf{a_j}$ est lu et donc reconnu, que la tête d'entrée pointe vers le symbole suivant de la bande d'entrée, que le sommet de la pile a été remplacé par la chaîne $\mathbf{x_n}$ (donc l'élément $\mathbf{x_n}$ a été empilé à la pile), que l'état de l'automate a changé et qu'il vaut maitenant $\mathbf{q_k}$, enfin que la tête de pile pointe sur le nouveau sommet :



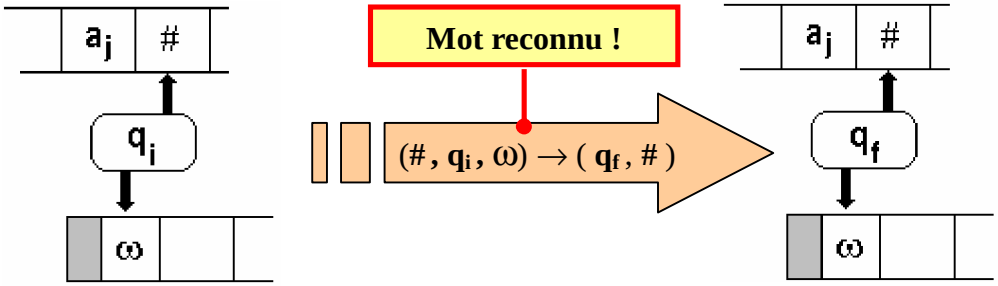
- **La transition $(a_j, q_i, \alpha) \rightarrow (q_k, \mathbf{Nil})$** signifie pour l'automate de ne rien faire dans la pile. Le résultat de la transition fait que l'état de l'automate passe de $\mathbf{q_i}$ à $\mathbf{q_k}$ et que la tête d'entrée pointe sur le symbole suivant de la bande d'entrée :



- **la transition $(a_j, q_i, \beta) \rightarrow (q_k, \#)$** signifie effacer l'actuel sommet de pile et pointer sur l'élément d'avant dans la pile (ce qui revient à dépiler la pile). Le résultat de la transition fait que l'état de l'automate passe de $\mathbf{q_i}$ à $\mathbf{q_k}$ et que la tête d'entrée pointe sur le symbole suivant de la bande d'entrée :



- Le mot est reconnu si l'automate rencontre une règle de transition de la forme **(# , q_i , ω)** → **(q_f , Nil)**, où **q_f** est un état final. L'automate s'arrête alors.



- Si la recherche de la transition μ : **(a_j , q_i ,** α) → .....ne donne pas de résultat on dit que l'auomate bloque : le mot n'est pas reconnu.

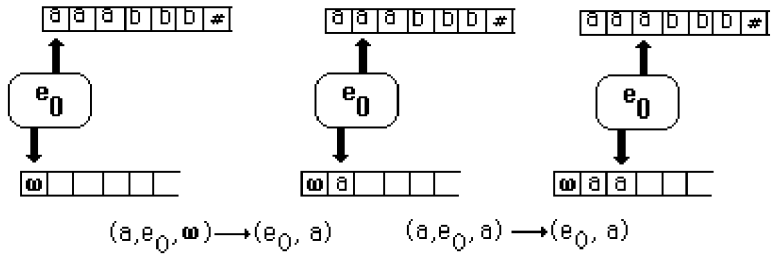
Exemple :

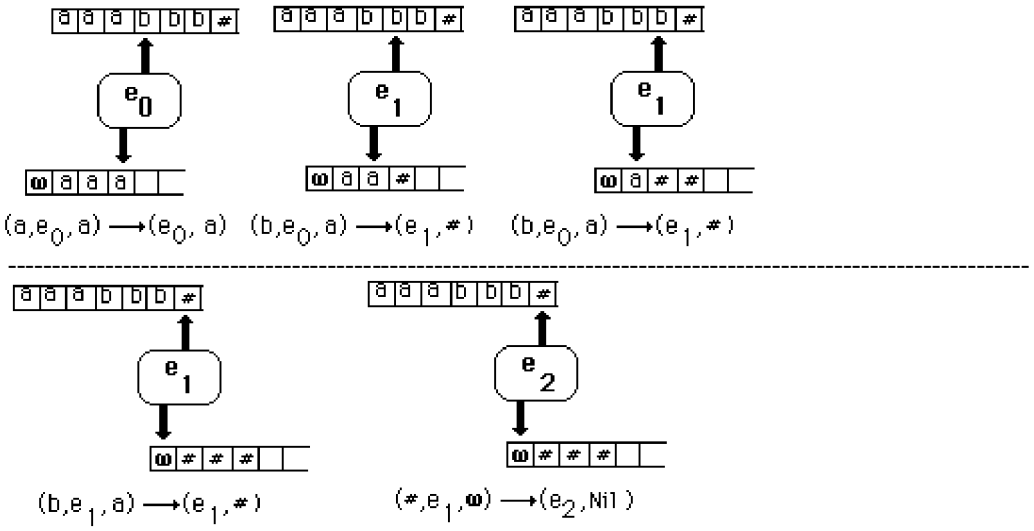
E= {**e0,e1,e2**}
e0 ∈ E (**e0** , état initial)
F ={**e2**} (**F**, état final **e2**)
$\mathbf{V_T}$ = {a,b,#}
$\mathbf{V_p}$ = {a,b,#,ω,**Nil**}

Règles de transitions:

(a , e0, ω) → (**e0**, **a**)
(a , e0, a) → (**e0**, **a**)
(b , e0, a) → (**e1**, #)
(b , e1, a) → (**e1**, #)
(# , e1, ω) → (**e2**, **Nil**)

Fonctionnement sur un exemple: reconnaissance du mot **aaabbb** :





Propriété :

> Un langage est un C-langage (engendré par une C-grammaire) **ssi** il est accepté par un automate à pile.

L'automate précédent reconnaît le C-langage L suivant :

L = {$\mathbf{a}^n\mathbf{b}^n$}(**a...ab...b**,n symboles **a** concaténés à n symboles **b**, n ≤ 1) sur l'alphabet V_T={**a**,**b**} dont une C-grammaire G est :

V_T = { **a** , **b** }
V_N = { S , A }
Axiome: S
Règles :
1 : S → **aSb**
2 : S → **ab**

2.3 Programme C# d'une classe d'automate à pile

Nous utilisons une classe de pile Lifo **ClassLifo** événementielle, déjà définie auparavant pour représenter la pile de l'automate. Afin que la pile Lifo de l'automate gère facilement des éléments de type **string,** nous proposons de la dériver du type **Stack** générique qui est une classe de pile générique :

```
//--> déclaration du type délégué :événement normalisé
// avec  information
public delegate void EventLifoHandler(object sender, EventArgs e);

class classLifo <T> : Stack<T>
{
  public event EventLifoHandler OnEmpiler;
  public event EventLifoHandler OnDepiler;

public bool EstVide ( )  { … }
public void EffacerPile ( ) { … }
public void Empiler (T x) { … }
public void Depiler (out T somm) { … }
public T Sommet { … }
public  void AfficherPile () { … }

}
```

Nous construisons une classe abstraite d'automate à pile **AutomateAbstr** qui implante toutes fonctionnalités d'un automate à pile, mais garde abstraite et virtuelle la méthode **transition** qui contient les règles de transitions de l'automate, ceci afin de déléguer son implementation à chaque classe d'automate particulier. Chaque classe fille héritant de **AutomateAbstr** redéfinira la méthode virtuelle **transition**.

Signatures des classes :

```
public class Lifo<T> : Stack<T>
{
    public bool EstVide()
    public void EffacerPile()
    public void Empiler(T x)
    public void Depiler(out T somm)
    public void AfficherPile()
}
```

```
public abstract class AutomateAbstr
{
    protected string FMot;
    protected Lifo<char> pile;
    protected int EtatFinal;
    protected const char omega = '$';
    protected const int non = -1;
    protected const char kNil = '@';
    protected const char sentinelle = '#';

    protected void init_Pile()
    protected abstract void transition(char ai, int qj, char alpha,
                                       out int qk, out char beta);
    public virtual string Mot
    public void Analyser()
}
```

Agrégation forte

La méthode abstraite virtuelle dans la classe mère.

```
public class AutomatePile : AutomateAbstr
{
    public AutomatePile(int fin)
    protected override void transition(char ai, int qj, char alpha,
                                       out int qk, out char beta)
}
```

La méthode abstraite redéfinie dans la classe fille.

Code de la classe de l'automate à pile : la pile Lifo générique

```
using System;
using System.Collections.Generic;
using System.Text;

// pile Lifo générique simple, héritant de Stack générique
namespace cci
{
  //--> classe mère des exceptions de Lifo
  public class LifoException : Exception
  {
    private string classObjEmetteur = "pas d'information sur l'objet émetteur.";

    //- propriété renvoyant la classe de l'émetteur de l'exception :
    public string classeObj
```

```
    {
      get   {  return classObjEmetteur;  }
    }

    //- constructeurs de l'exception LifoException :
    public LifoException(string mess): base(mess) {
    }

    public LifoException(string mess, object x): base(mess)    {
      Type TypeEmetteur = x.GetType();
      classObjEmetteur = TypeEmetteur.FullName +
                    " dans " + TypeEmetteur.Assembly.GetName();
    }
  }
  //--> classe fille d'exception Lifo de pile vide :
  public class LifoEmptyException : LifoException
  {
    //- constructeur LifoEmptyException:
    public LifoEmptyException(string mess): base("Pile vide : " + mess)
    {
    }
  }

  class LifoStack<T> : Stack<T>
  {
    //- méthode indiquant si la pile est vide :
    public bool EstVide()
    {
      return this.Count == 0;
    }

//- méthode vidant entièrement la pile :
    public void EffacerPile()
    {
      this.Clear();
    }

//- méthode surchargée empilant un objet :
    public void Empiler(T x)
    {
      this.Push(x);
    }

    //- méthode dépilant un objet :
    public void Depiler(out T somm)
    {
      somm = default(T);
      if (!EstVide())
      {
        somm = this.Pop();
      }
      else
        throw new LifoEmptyException("impossible de dépiler");
    }

    /* propriété renvoyant le sommet de la pile :
     *  lève une exception si la pile est vide
    */
    public T Sommet
    {
      get
```

```
      {
        if (!EstVide())
          return this.Peek();
        else
          throw new LifoEmptyException("sommet inexistant");
      }
    }

//- méthode affichant entièrement la pile :
    public void AfficherPile()
    {
      T[] tableau = this.ToArray();
      for (int i = 0; i < tableau.Length; i++)
        System.Console.Write(tableau[i] + " ");
      System.Console.WriteLine();
    }
  }
}
```

Code complet des classes AutomateAbstr et AutomatePile

```
using System;
using System.Collections.Generic;
using System.Text;

namespace cci
{
 public abstract class AutomateAbstr
 {
  protected string FMot;
  protected Lifo<char> pile;
  protected int EtatFinal;
  protected const char omega = '$';
  protected const int non = -1;
  protected const char kNil = '@';
  protected const char sentinelle = '#';

  protected void init_Pile()
  {
   pile.EffacerPile();
   pile.Empiler(omega);
  }

  protected abstract void transition(char ai, int qj,
                                     char alpha, out int qk, out char beta);

  public virtual string Mot
  {
   get
   {
    return FMot;
   }
   set
   {
    int Long = value.Length;
    if (Long != 0)
    {
     if (value[Long - 1] != sentinelle)
      FMot = value + sentinelle;
     else   FMot = value ;
```

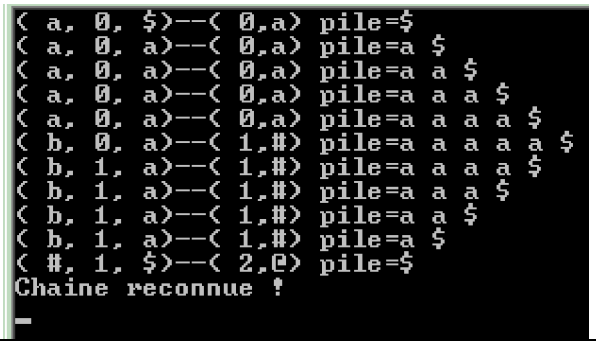
```
    }
    else
     FMot = sentinelle.ToString();
   }
  }

  public void Analyser()
  {
   int q = 0;// état initial q0 = 0
   int numcar = 0;// les string débutent à 0
   char carpile = (char)0;// pour inialisation
   char s = (char)0;// pour inialisation

   init_Pile();
   while (q != non & q != EtatFinal)
   {
    transition(FMot[numcar], q, pile.Sommet, out q, out carpile);
    pile.AfficherPile();
    numcar++;
    if (carpile == sentinelle)
     pile.Depiler(out s);
    else
     if (carpile == 'a' | carpile == 'b')
      pile.Empiler(carpile);
   }
   if (q == EtatFinal & carpile == kNil)
    Console.WriteLine("Chaîne reconnue.");
   else
    Console.WriteLine("Blocage, chaîne non reconnue.");
  }
 }

/* ----------------    classe Automate à pile   --------------- */
 public class AutomatePile : AutomateAbstr
 {
  //-- constructeur :
  public AutomatePile(int fin)
  {
   pile = new Lifo<char>();
   this.init_Pile();
   if (fin >= 1 & fin <= 20)
    EtatFinal = fin;
   else
    EtatFinal = 20;
   FMot = sentinelle.ToString();
  }
  protected override void transition(char ai, int qj,
                                     char alpha, out int qk, out char beta)
  {
   Console.Write("(" + ai + "," + qj + "," + alpha + ")");
   if (ai == 'a' & qj == 0 & alpha == omega)
   {
    qk = 0;
    beta = 'a';
   }
   else
    if (ai == 'a' & qj == 0 & alpha == 'a')
    {
     qk = 0;
     beta = 'a';
```

```
    }
    else
     if (ai == 'b' & qj == 0 & alpha == 'a')
     {
      qk = 1;
      beta = sentinelle;
     }
     else
      if (ai == 'b' & qj == 1 & alpha == 'a')
      {
       qk = 1;
       beta = sentinelle;
      }
      else
       if (ai == sentinelle & qj == 1 & alpha == omega)
       {
        qk = EtatFinal;
        beta = kNil;
       }
       else // blocage dans tous les autres cas
       {
        qk = non;
        beta = kNil;
       }
   Console.Write("--(" + qk + "," + beta + ") pile = ");
  }
 }
}
```

Programme utilisant la classe Automate

```
namespace cci {
 class Program
 {
  static void Main(string[] args)
  {
   AutomatePile Automate = new AutomatePile(2);
   Automate.Mot = "aaaaabbbbb";
   Automate.Analyser();
   Console.ReadLine();
  }
 }
}
```

Exécution de ce programme sur l'exemple ***aaaaabbbbb*** *:*



La construction générale et systématique de tous ces automates à pile dépasse le cadre de ce document, il est conseillé de poursuivre dans les ouvrages signalés dans la bibliographie.

14.2 : Automates et grammaires de type 3

Plan du chapitre:

1. Automates d'états finis pour les grammaires de type 3

Dans ce chapitre, le point de vue adopté est celui de l'implantation pratique des notions proposées en C#. La reconnaissance automatique et méthodique est très aisément accessible dans le cas des grammaires de type 3 à travers les automates d'états finis. Nous fournissons les éléments théoriques appuyés sur des exemples pratiques.

1.1 Automates d'états finis déterministes ou non

Définition

C'est un Quintuplet $A = (V_T, E, q_0, F, \mu)$ où :
V_T : **vocabulaire terminal** de A.
E : **ensemble des états** de A ; $E = \{q_0, q_1, ..., q_n\}$
$q_0 \in E$ est dénommé **état initial** de A.
$F \subset E$: F est l'ensemble des **états finaux** de A.
$\mu : E \times V_T \rightarrow E$, une **fonction de transition** de A.

Définition

Un automate A, $A = (V_T, E, q_0, F, \mu)$, est dit **déterministe**, si sa fonction de transition μ est une vraie fonction au sens mathématique. Ce qui revient à dire qu'un couple de $E \times V_T$ n'a qu'**une seule image** par μ dans E.

Intérêt de la notion d'automate d'états finis

Comme pour les automates à pile, c'est la fonction de transition qui est l'élément central de l'automate A. Elle doit être définie de manière à permettre d'analyser un mot de V_T^*, et aussi de décider de l'appartenance ou non d'un mot à un certain langage. Ce langage d'appartenance est appelé le langage reconnu par l'automate.

Exemple :

Soit un automate possédant parmi ses règles, les trois suivantes : $(q_i, a_j) \rightarrow q_k^1$ $(q_i, a_j) \rightarrow q_k^2$ ..... $(q_i, a_j) \rightarrow q_k^n$	Il existe trois règles ayant la même partie gauche (q_i, a_j), ce qui revient à dire que le couple (q_i, a_j) a trois images distinctes, donc l'automate est **non déterministe**.

Par la suite, pour des raisons de simplification pratique, nous considérerons les **A**utomates d'**E**tats **F**inis normalisés que nous nommerons AEF, en posant :

F = { q_f } un seul état final
$V_T = V_T \cup \{ \# \}$ on ajoute dans V_T un symbole terminal de fin de mot #, le même pour tous les AEF.

1.2 Fonctionnement pratique d'un AEF :

Nous construisons notre AEF comme étant un dispositif physique muni :

- d'une **bande d'entrée** (de papier, ou magnétique par exemple) composée de cases ne pouvant contenir chacune qu'un seul symbole de $\mathbf{V_T}$ à la fois,
- d'une seule tête de lecture de symobles capable de reconnaître des éléments du vocabulaire terminal $\mathbf{V_T}$ dans chaque case de la **bande d'entrée**, et **possédant plusieurs états**. La tête de lecture se déplace de gauche à droite d'une case à la fois,

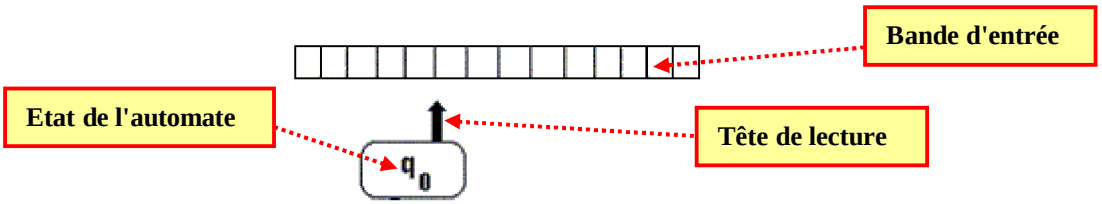
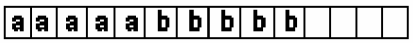


fig - un automate d'états finis

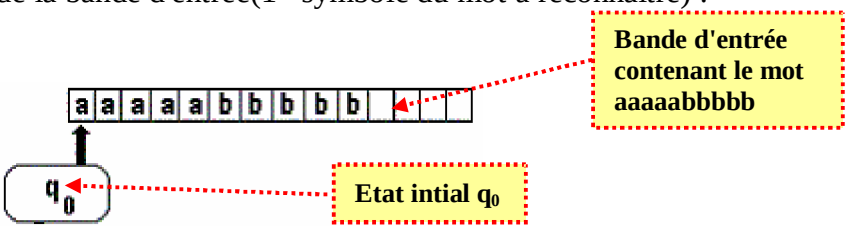
- Les règles de transitions spécifient la manipulation de la bande d'entrée de l'automate.

L'algorithme de fonctionnement d'un AEF :

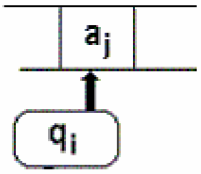
L'algorithme est très semblable à celui d'un automate à pile (en fait on peut considérer qu'il s'agit d'un cas particulier d'automate à pile dans lequel on n'effectue jamais d'action dans la pile), on fournit un mot que l'on écrit symbole par symbole de gauche à droite dans chaque case de l'automate (par exemple avec $\mathbf{V_T}$ ={a,b} le mot **aaaaabbbbb**):



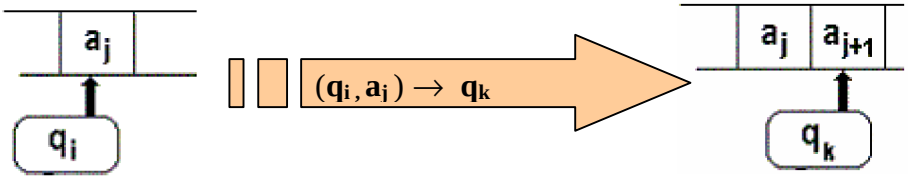
L'automate est mis à l'état initial $\mathbf{q_0}$, sa tête de lecture d'entrée est positionnée sur la première case à gauche de la bande d'entrée(1er symbole du mot à reconnaître) :



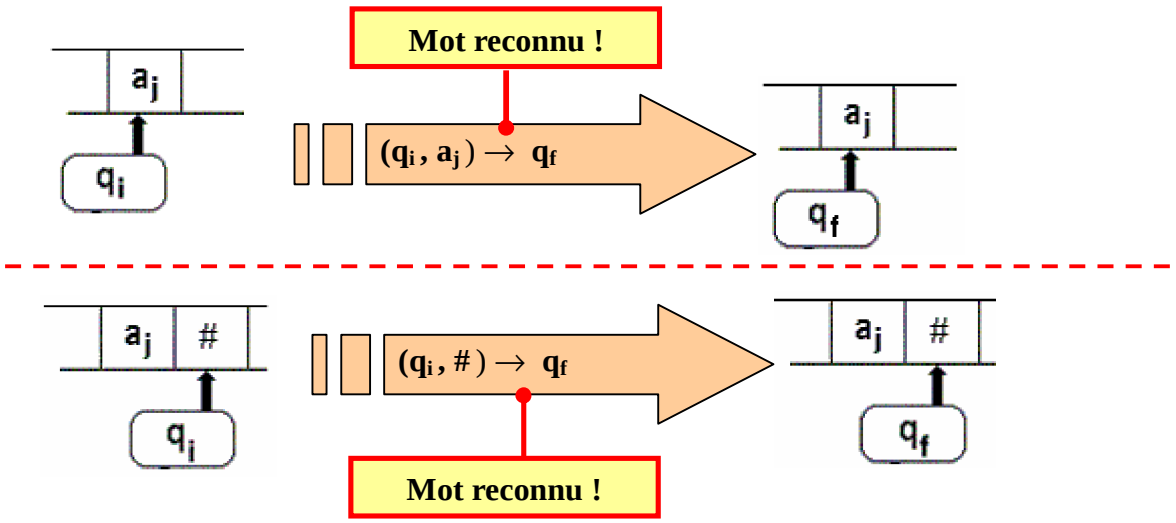
- La tête de lecture se déplace par examen des règles de transition de. Le couple ($\mathbf{a_j}$, $\mathbf{q_i}$) enclenche le processus de recherche d'une transition possible dans la partie gauche de la liste des règles de transitions de μ (il y a recherche de la transition μ : ($\mathbf{a_j}$, $\mathbf{q_i}$) → .....).

Supposons que la case contienne le symbole a_j que la tête soit à l'état q_i	

- **La transition** $(q_i, a_j) \rightarrow q_k$ signifie que l'automate peut passer de l'état q_i à l'état q_k à condition que le mot d'entrée débute par la chaîne préfixe a_j élément de V_T^* (notons que la chaîne a_j peut être réduite par sa définition à un seul élément de V_T , ce qui est généralement le cas pratique. Le résultat de la transition fait que le symbole a_j est lu et donc reconnu, que la tête d'entrée pointe vers le symbole suivant de la bande d'entrée :



- Le mot est reconnu si l'automate rencontre une règle de transition de la forme $(q_i, a_j) \rightarrow q_f$ ou bien $(q_i, \#) \rightarrow q_f$ où q_f est un état final. L'automate s'arrête alors.



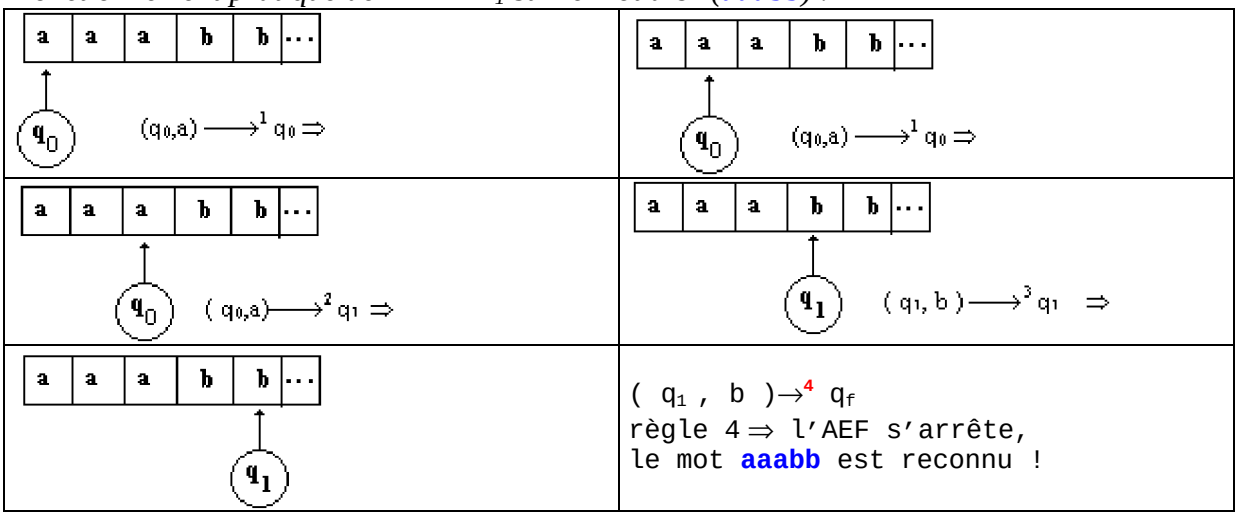
- Si l'AEF ne trouve pas de règle de transition commençant par (q_j , a_i), c'est à dire que le couple (q_j , a_i) n'a pas d'image par la fonction de transition µ, on dit alors que l'automate **bloque** : le mot n'est pas reconnu comme appartenant au langage.

1.3 Utilisation d'un AEF en reconnaissance de mots

Soit la grammaire G_1 déjà étudiée précédemment dont le langage engendré est celui des mots de la forme $a^n b^p$.

grammaire G_1	Soit l'automate A_1 :
$L(G_1) = \{ a^n b^p / (n \leq 1) \text{ et } (p \leq 1) \}$ $G_1 : V_N = \{S,A\}$ $V_T = \{a,b\}$ **Axiome** : S **Règles** 1 **:** S → aS 2 : S → aA 3 : A → bA 4 : A → b	$V_T = \{a,b\}$ $E = \{q_0 , q_1 , qf \}$ $\mu : (q_0, a) \rightarrow q_0$ $\mu : (q_0, a) \rightarrow q_1$ $\mu : (q_1, b) \rightarrow q_1$ $\mu : (q_1, b) \rightarrow q_f$ *(A_1 est non déterministe)*

*Fonctionnement pratique de l'AEF A_1 sur le mot a^3b^2 (**aaabb**) :*



Nous remarquons que dans le cas de cet AEF, il nous a fallu aller " voir " un symbole plus loin, afin de déterminer la bonne règle de transition pour mener jusqu'au bout l'analyse.

On peut montrer que tout AEF non déterministe peut se ramener à un AEF déterministe équivalent. Nous admettons ce résultat et c'est pourquoi nous ne considérerons par la suite que les AEF déterministes (notés **AEFD**).

Voici à titre d'exemple un AEFD équivalent à l'AEF A_1 précédent :

Soit l'AEFD A_2 reconnaissant le langage $L=\{ a^n b^p / (n \leq 1) \text{ et } (p \leq 1) \}$, nous aurons donc deux automates reconnaissant le langage L(l'un est déterministe, l'autre est non déterministe), ci-dessous un tableau comparatif de ces deux automates d'états finis :

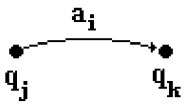
AEF A_1 non déterministe	AEF A_2 déterministe
$V_T = \{a,b\}$ $E = \{q_0 , q_1 , qf \}$ $\mu : (q_0, a) \rightarrow q_0$ $\mu : (q_0, a) \rightarrow q_1$ $\mu : (q_1, b) \rightarrow q_1$	$V_T = \{a,b,\#\}$ $E = \{q_0,q_1, q_2,qf \}$ $\mu : (q_0, a) \rightarrow q_1$ $\mu : (q_2, b) \rightarrow q_2$ $\mu : (q_1, a) \rightarrow q_1$

$\mu : (q_1, b) \rightarrow q_f$	$\mu : (q_1, b) \rightarrow q_2$ $\mu : (q_2, \#) \rightarrow q_f$

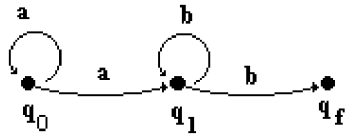
Voici la reconnaissance automatique du mot $\mathbf{a^3b^2}$ par l'automate AEFD A_2 :
$(q_0, \mathbf{a}) \rightarrow q_1$
$(q_1, \mathbf{a}) \rightarrow q_1$
$(q_1, \mathbf{a}) \rightarrow q_1$
$(q_1, \mathbf{b}) \rightarrow q_2$
$(q_2, \mathbf{b}) \rightarrow q_2$
$(q_2, \#) \rightarrow q_f$ mot reconnu !

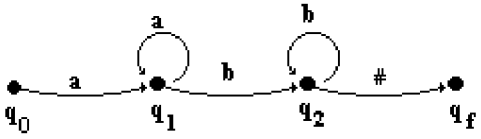
1.4 Graphe d'un automate déterministe ou non

C'est un graphe orienté, représentant la suite des transitions de l'automate comme suit :

$(q_j , a_i) \rightarrow q_k$ est représentée par l'arc à 2 sommets : 

Exemples du graphe des deux AEF précédents :

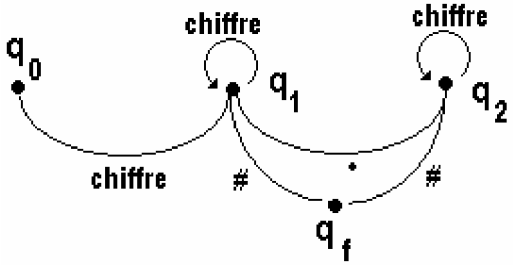
graphe de A_1 : 

graphe de A_2 : 

Que l'AEF soit déterministe ou non, il est toujours possible de lui associer un tel graphe.

Exemple : reconnaître des chiffres

Voici un **A**utomate d'**E**tats **F**inis **D**éterministe qui reconnaît :
- les constantes chiffrées terminées par un #,

AEFD représenté par son graphe	AEFD représenté par ses règles
	$(q_0 , \mathbf{chiffre}) \rightarrow q_1$ $(q_1, \mathbf{chiffre}) \rightarrow q_1$ $(q_1 , \#) \rightarrow q_f$ $(q_1 , .) \rightarrow q_2$ $(q_2 , \mathbf{chiffre}) \rightarrow q_2$ $(q_2 , \#) \rightarrow q_f$ où $(q_0 , \mathbf{chiffre})$ représente un notation pour les 10 couples : $(q_0 , \mathbf{0})$,…, $(q_0 , \mathbf{9})$

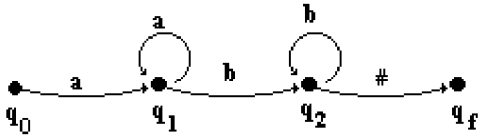
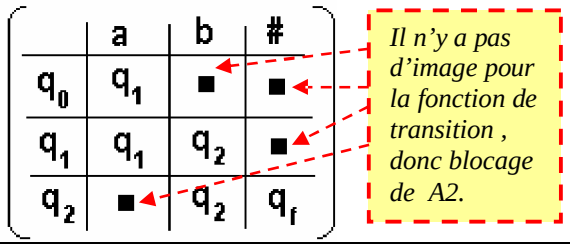
On peut voir à travers ce dernier exemple, le schéma général d'un analyseur lexical qui constitue la première étape d'un compilateur. Il suffit dès que le symbole a été reconnu donc à partir d'un état final $\mathbf{q_f}$ de repartir à l'état q_0.

1.5 Matrice de transition : dans le cas déterministe

On représente la fonction de transition par une matrice **M** dont les cellules sont toutes des états de l'AEF (ensemble E), où les colonnes contiennent les éléments de V_T (symboles terminaux), les lignes contiennent les états (éléments de E sauf l'état final q_f).

La règle (q_j , a_i) → q_k est stockée de la façon suivante M(i,j)= q_k où :
la ligne j correspond à l'état q_j
la colonne i correspond au symbole a_i

Exemple : la matrice des transitions de A_2

AEFD A2 représenté par son graphe	AEFD A2 représenté par sa matrice
	

Utilisation pratique de la matrice des transitions

Dénommons Mat[i,j] l'état valide de coordonnées (i,j) dans la matrice des transitions d'un AEFD. Un schéma d'algorithme de reconnaissance par l'AEFD est très simple à décrire :

```
Etat ← q0 ;
Symlu ← premier symbole du mot ;
tantque Etat ≠ qf Faire
 Etat ←Mat[Etat,Symlu] ;
 Symlu ←Symbole suivant
Fintant ;
```

2. Automates et grammaires de type 3

Il existe une correspondance bijective entre les K-grammaires (**grammaires de type-3**) et les **AEF**. Cette correspondance est la base sur laquelle nous systématisons l'implantation d'un AEFD. En voici une construction pratique.

2.1 Automate associé à une K-grammaire

Soit G une K-grammaire , G = (V_N, V_T, S, R)
On cherche l'AEF A, A = (V_T', E, q_0, q_f, μ), tel que A reconnaisse G.

Soit la construction suivante de l'AEF A :

Grammaire G	AEF A associé
V_T	$V_T' = V_T \cup \{\#\}$
Chaque élément de V_N est un q_j de E	$E = V_N \cup \{q_f\}$
A toute règle terminale de G de la forme $A_j \rightarrow a_k$	on associe la règle de transition $(q_j, a_k) \rightarrow q_f$
S est l'axiome de G	$q_0 = S$
A toute règle non terminale de G de la forme $A_j \rightarrow a_k A_i$	on associe la règle de transition $(q_j, a_k) \rightarrow q_i$

L'automate A ainsi construit reconnaît le langage engendré par G.

Remarque :

L'automate A_1 reconnaissant le langage $\{a^n b^p / (\mathbf{n} \leq 1) et (\mathbf{p} \leq 1)\}$ associé à la grammaire G_1 (cf. plus haut) a été construit selon cette méthode.

Exemple : soit G une K-grammaire suivante et **Aut** l'automate associé par le procédé bijectif précédent

G K-grammaire	**Aut** automate associé
G = (V_N, V_T, S, R) V_T = { a, b, c, # } V_N = { S, A, B } **Axiome** : S **Règles** de G : 1 : S → aS 2 : S → bA 3 : A → bB 4 : B → cB 5 : B → #	**Aut** = (V_T', E, q_0, q_f, μ) $V_T' = V_T$ S est associé à : q_0 A est associé à : q_1 B est associé à : q_f 1 : S → aS est associé à :(q_0, a) → q_0 2 : S → bA est associé à :(q_0, b) → q_1 3 : A → bB est associé à :(q_1, b) → q_2 4 : B → cB est associé à :(q_2, c) → q_2 5 : B → # est associé à :(q_2, #) → q_f

Etudions maintenant la construction réciproque d'une K-grammaire à partir d'un AEF.

2.2 Grammaire associée à un Automate

Soit l'AEF **Aut**, tel que A = (V_T', E , q_0 , q_f ,μ)
On cherche G = (V_N,V_T,S,R) une K-grammaire du langage reconnu par cet automate.

AEF A ut	**Grammaire G associée**
V_T'	$V_T = V_T'$
E	V_N = E -{q_f}
q_0	**Axiome** : q_0
si $q_j \neq q_f$ **alors** pour (q_j, a_k)→ q_i	on construit : [r : $q_j \rightarrow a_k q_i$] dans G
si $q_j = q_f$ **alors** pour (q_j, a_k) → q_f	on construit : [r : $q_j \rightarrow a_k$] dans G

Exemple : soit l'automate Aut reconnaissant le langage {$a^n b^2 c^p$ / n ≤ 0 et p ≤ 0} et G une grammaire associée

Aut automate	G K-grammaire associée
V_T = { a, b, c, # } E = {q_0,q_1, q_2,qf } **transitions :** (1) (q_0, a) → q_0 *[les a^n]* (2) (q_0, b) → q_1 (3) (q_1, b) → q_2 *[le b^2]* (4) (q_2, c) → q_2 *[les c^p]* (5) (q_2, #) → q_f	S remplace q_0 On pose : V_T = { a, b, c, # } A remplace q_1 On pose : V_N = { S, A, B } B remplace q_2 **Axiome** : S **règles :** 1 : S → aS remplace (q_0, a) → q_0 2 : S → bA remplace (q_0, b) → q_1 3: A → bB remplace (q_1, b) → q_2 4 : B → cB remplace (q_2, c) → q_2 5 : B → # remplace (q_2, #) → q_f

La grammaire **G** de type 3 associée par la méthode précédente est celle que nous avons déjà vue dans l'exemple précédent.

3. Implantation d'une classe d'AEFD en C#

Nous proposons deux constructions différentes de la fonction de transition d'un AEFD soit en utilisant directement les règles de transition, soit à partir de la matrice de transition.

Nous construisons comme dans le paragraphe précédent un squelette de classe abstraite d'automate d'état fini :

```
public abstract class AEFDAbstr
{
      protected string FMot;
      protected int EtatFinal;
      protected const int non = -1;
      protected const int fin = 20;
      protected const char sentinelle = '#';
      protected abstract int transition(int q, char car);

      protected char Symsuiv(int n) {…}
      public virtual string Mot   {…}
      public void Analyser(){…}
  }
```

La méthode abstraite virtuelle dans la classe mère.

Cette classe se décline en deux versions selon que l'on implante l'automate par ses règles de transition ou bien par sa matrice de transition.

3.1 Fonction de transition à partir des règles

Méthodologie

Toute règle est de la forme $(q_i, a_k) \rightarrow q_i$, donc nous pourrons prendre comme modèle d'implantation de la fonction de transition une **méthode function** d'une classe que nous nommerons AEFD, dont voici ci-dessous une écriture fictive pour une seule règle.

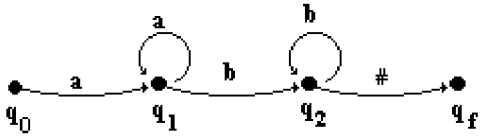
```
int transition (int  q , char car)
{
  if (q == qi & car == ak) q = qi ;  {la règle (qi, ak) → qi}
  else ....
}
```

Toutes les autres règles sont implantées dans la **méthode** transition de la même façon.

L'appel de la méthode transition a lieu dans la méthode Analyser:

```
public void Analyser()
{
   int numcar = 0;
   int etat = 1;
   char symLu ;
   while (etat != non & etat != fin) do
   {
     symLu = Symsuiv(numcar++) ; { fournit dans symLu le symbole suivant }
     etat = transition(etat, symLu);
   }
   if (etat == EtatFinal)
     Console.WriteLine("Chaîne reconnue.");
   else
     Console.WriteLine("Blocage, chaîne non reconnue.");
}
```

Exemple : morceaux de code par règles de transition sur un automate déjà étudié

la méthode transition de l'automate déterministe A_2 : (reconnaissant le langage $a^n b^p$)	


```
protected int EtatFinal;
protected const int non = -1;
protected const int fin = 20;
protected const char sentinelle = '#';

int transition (int q, char car)
{    // valeurs choisies : q0 = 1, q1 = 2, q2 = 3, qf = EtatFinal
        Console.Write("(" + car + "," + q + ")");
        if (q == 1 & car == 'a')
           q = 2; //(q0,a) --> q1
        else
           if (q == 2 & car == 'a')
              q = 2; //(q1,a) -->q1
           else
              if (q == 2 & car == 'b')
                 q = 3; //(q1,b) --> q2
              else
                 if (q == 3 & car == 'b')
                    q = 3; //(q2,b) --> q2
                 else
                    if (q == 3 & car == sentinelle)
                       q = EtatFinal; //(q2,#) --> qf}
                    else
                       q = non; //blocage, le caractère n'est pas dans Vt
        return q;
}
```

Méthode de **transition** selon les régles.

Comme l'automate est déterministe, il est possible de procéder différemment en utilisant sa matrice de transition, ce qui est un bon exemple d'application d'utilisation de la notion de matrice dans un programme.

3.2 Fonction de transition à partir de la matrice

Méthodologie

Une autre écriture d'un même AEFD est obtenue (lorsque cela est possible en place mémoire) à partir d'un **tableau** C# (dénoté **table**) représentant la matrice des transitions de l'automate. Nous utilisons le même modèle d'implantation de la fonction de transition que dans le cas d'une description par règles (**méthode function** d'une classe AEFD).

Version réduite de la classe abstraite d'AEFD :

```
public abstract class AEFDAbstr
{
    const int non = 0;
    const int fin = 20;
    const char sentinelle = '#';
    .....
    int[ , ] table = new int[ fin , 256 ];
    .....
}
```

Il est nécessaire d'initialiser la matrice table avec les valeurs de départ de l'AEFD. Une méthode **init_table** pourra se charger de ce travail. Dans ce cas, la **méthode transition** est très simple à écrire, elle se résume à parcourir la matrice des transitions accessible comme champ de la classe :

Version augmentée de la classe abstraite d'AEFD avec la méthode init_table abstraite :

```
public abstract class AEFDAbstr
{
      protected string FMot;
      protected int EtatFinal;
      protected const int non = 0;
      protected const int fin = 20;
      protected const char sentinelle = '#';

      protected int[,] table = new int[fin ,256];
      protected abstract void init_table();

      protected abstract int transition(int q, char car);
      protected char Symsuiv(int n) {…}
      public virtual string Mot   {…}
      public void Analyser(){…}
  }
```

L'appel de la **méthode** transition se fait comme dans le cas précédent, à travers un objet AEFD de classe AutomateEF :

Exemple : l'automate reconnaissant le langage $a^n b^p$
Nous reprenons l'automate du paragraphe précédent, mais en l'implantant grâce à sa table de transition.

morceaux de code

la méthode transition de l'automate déterministe A_2 :
(reconnaissant le langage $a^n b^p$)

	a	b	#
q_0	q_1	■	■
q_1	q_1	q_2	■
q_2	■	q_2	q_f

Implantation de la fonction de transition et de la table dans cet exemple :

```
protected override int transition(int q, char car)
{
        q = table[q, car];
        return q;
}
```

Méthode de **transition** selon la matrice de transition.

```
protected override void init_table()
{ //par défaut tout est non reconnu :
        for (int i = non; i < fin; i++)
                for (int j = 0; j < 256; j++)
                  table[i, j] = non;
// Règles spécifiques de l'AEFD a^n b^p
// Valeurs : q0 = 1, q1 = 2, q2 = 3, qf = EtatFinal
    table[1, 'a'] = 2;
    table[2, 'a'] = 2;
    table[2, 'b'] = 3;
    table[3, 'b'] = 3;
    table[3, sentinelle] = EtatFinal;
}
```

Code complet des classes AutomateAbstr et AEFD

```
using System;
using System.Collections.Generic;
using System.Text;

namespace cci
{
 public abstract class AEFDAbstr
 {
  protected string FMot;
  protected int EtatFinal;
  protected const int non = 0;
  protected const int fin = 20;
  protected const char sentinelle = '#';
  protected int[,] table = new int[fin ,256];

  protected abstract int transition(int q, char car);
  protected abstract void init_table();

  protected char Symsuiv(int n)
  {
   return FMot[n];
  }

  public virtual string Mot
  {
   get {    return FMot;    }
   set {
          int Long = value.Length;
          if (Long != 0)
          {
           if (value[Long - 1] != sentinelle)
            FMot = value + sentinelle;
           else
             FMot = value ;
          }
          else
           FMot = sentinelle.ToString();
       }
  }

  public void Analyser()
```

```
  {
   int etat = 1;// état initial q0 = 1
   int numcar = 0;// les string débutent à 0
   char symLu = (char)0;
   while (etat != non & etat != EtatFinal)
   {
    symLu = Symsuiv(numcar++);
    etat = transition(etat, symLu);
   }
   if (etat == EtatFinal)
    Console.WriteLine("Chaîne reconnue.");
   else
    Console.WriteLine("Blocage, chaîne non reconnue.");
  }
 }



public class AEFD : AEFDAbstr
 {
  //-- constructeur :
  public AEFD(int end)
  {
   if (end >= 2 & end <= fin)
    EtatFinal = end;
   else
    EtatFinal = fin;
   FMot = sentinelle.ToString();
   init_table();
  }
  protected override void init_table()
  {   //par défaut tout est non reconnu:
   for (int i = non; i < fin; i++)
    for (int j = 0; j < 256; j++)
     table[i, j] = non;
   //Les règles de l'AEFD : q0 = 1, q1 = 2, q2 = 3, qf = EtatFinal
   table[1, 'a'] = 2;
   table[2, 'a'] = 2;
   table[2, 'b'] = 3;
   table[3, 'b'] = 3;
   table[3, sentinelle] = EtatFinal;

  }
  protected override int transition(int q, char car)
  {
       // par règles de transition ou par matrice de transition
       // voir ci-dessous . . .
  }
}
```

// implantation C# de la méthode « **override** int transition » par les règles de transition

```
protected override int transition(int q, char car)
  {
   // valeurs choisies : q0 = 1, q1 = 2, q2 = 3, qf = EtatFinal
   Console.Write("(" + car + "," + q + ")");
   if (q == 1 & car == 'a')
    q = 2; //(q0,a) --> q1
   else
    if (q == 2 & car == 'a')
     q = 2; //(q1,a) -->q1
```

```
    else
     if (q == 2 & car == 'b')
      q = 3; //(q1,b) --> q2
     else
      if (q == 3 & car == 'b')
       q = 3; //(q2,b) --> q2
      else
       if (q == 3 & car == sentinelle)
        q = EtatFinal; //(q2,#) --> qf}
       else
        q = non; //blocage, le caractère n'est pas dans Vt
   Console.WriteLine("--> " + q);
   return q;
  }
```

// implantation C# de la méthode « **override** int transition » par matrice de transition

```
protected override int transition(int q, char car )
{
   Console.Write("(" + car + "," + q + ")");
   q = table[q, car];
   Console.WriteLine("--> " + q);
   return q;
  }
}
```

Programme utilisant la classe AutomateEF

Reconnaissant le langage L = { $a^n b^p$ / (n ≤ 1) et (p ≤ 1) }

```
class ProgramAEFD
{
 static void Main(string[] args)
 {
    AEFD Automate = new AEFD(4);
    Automate.Mot = "aaaaabbbbb";
    Automate.Analyser();
    Console.ReadLine();
 }
}
```

*Exécution de ce programme sur l'exemple **aaaaabbbbb** :*

```
(a,1)--> 2
(a,2)--> 2
(a,2)--> 2
(a,2)--> 2
(a,2)--> 2
(b,2)--> 3
(b,3)--> 3
(b,3)--> 3
(b,3)--> 3
(b,3)--> 3
(#,3)--> 4
Chaîne reconnue.
```

*Exécution de ce programme sur l'exemple **aaaaabbabb** :*

```
(a,1)--> 2
(a,2)--> 2
(a,2)--> 2
(a,2)--> 2
(a,2)--> 2
(b,2)--> 3
(b,3)--> 3
(a,3)--> 0
Blocage, chaîne non reconnue.
```

Nous remarquons dans ce dernier paragraphe, que nous avons mis en place un procédé général qui permet de construire méthodiquement en C# une classe d'AEFD, **uniquement** en changeant le contenu de la méthode **init_table**. Nous avons en fait mis au point un générateur manuel de programme C# pour AEFD.

Par la suite, nous utiliserons ce procédé à chaque fois que nous aurons à programmer un AEFD. Nous n'aurons seulement qu'à déclarer une nouvelle classe héritant de AutomateAbstr et implémenter par redéfinition sa méthode **init_table** :

Nous terminons ce chapitre en détaillant complètement deux exercices de construction de programmes selon la méthode qui vient d'être décrite. Le lecteur pourra en inventer d'autres basés sur la même idée.

3.3 Exemple : les identificateurs C# - like

On donne des diagrammes syntaxiques de construction d'identificateurs genre C# :
identificateur :

Construisons un programme C# reconnaissant de tels identificateurs, en utilisant le procédé de génération manuelle avec la classe abstraite AutomateAbstr définie au paragraphe précédent.

Méthode de travail adoptée

- **Détermination d'une grammaire G adéquate.**
- **Construction de l'automate AEFD associé à G.**
- **Programme C# associé à l'automate construit.**

Détermination d'une grammaire G_{id} adéquate

Nous remarquons d'abord qu'il y a une grammaire de type 3 engendrant ces identificateurs :
Soient à considérer les ensembles :
Lettre = { **a**, **b**,..., **z** }. *// les 26 lettres de l'alphabet*
Chiffre = { **0**, **1**,..., **9** }. *// les 10 chiffres*
V_T = Chiffre ∪ Lettre ∪ {#}.
V_N = { ⟨*identificateur*⟩ , **A** }
Posons G_{id} = (V_T , V_N , ⟨*identificateur*⟩ , Règles) une grammaire où
Axiome : ⟨*identificateur*⟩

Règles :
⟨*identificateur*⟩ → **a |b|...|z A**
A → **a |b|...|z A**
A → **0 |1|...|9 A**
A → #
La grammaire G_{id} est de type 3 et déterministe.

Construction de l'automate associé à G_{id}

Afin de réduire le nombre de lignes de texte, nous adoptons les conventions d'écriture suivantes :

(q_k ,Lettre) → q_i , *représente l'ensemble des 26 règles :*
(q_k ,**a**) → q_i
.....
(q_k ,**z**) → q_i

(q_k , Chiffre) → q_i , *représente l'ensemble des 10 règles :*
(q_k ,**0**) → q_i
.....
(q_k ,**9**) → q_i

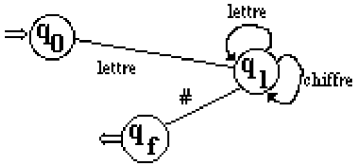
Etats associés aux éléments de V_N:
⟨*identificateur*⟩ ⇔ q_0
⟨A⟩ ⇔ q_1
Etat initial = q_0
Etat final = q_f

Vocabulaire terminal de l'automate :
$V_T' = V_T \cup \{\#\}$ = Chiffre ∪ Lettre ∪ {#}

Règles de l'automate associé à G_{id} :
(q_0 , Lettre) → q_1 *// 26 règles*
(q_1 , Lettre) → q_1 *// 26 règles*
(q_1 , Chiffre) → q_1 *// 10 règles*
(q_1 , #) → q_f

Soit un total de 63 règles.

Graphe de l'automate :



Matrice de transitions de l'automate:

	a	.....	z	0	...	9	#
q_0	q_1	.....	q_1	■	...	■	■
q_1	q_1	.....	q_1	q_1	...	q_1	q_f

Programme associé à l'automate

Nous héritons de la classe AutomateAbstr. Nous n'avons que la méthode **init_table** à redéfinir

```
public class AutomateIdentif : AEFDAbstr
{
      //-- constructeur :
      public AutomateIdentif (int end)
      {
         if (end >= 2 & end <= fin)
            EtatFinal = end;
         else
            EtatFinal = fin;
         FMot = sentinelle.ToString();
         init_table();
      }

    protected override int transition(int q, char car)
     // par matrice de transition
     {
              Console.Write("(" + car + "," + q + ")");
              q = table[q, car];
              Console.WriteLine("--> " + q);
              return q;
     }
```

```
protected override void init_table()
{   //par défaut tout est non reconnu:
          for (int i = non; i < fin; i++)
            for (int j = 0; j < 256; j++)
               table[i, j] = non;
          //Les règles de l'AEFD :
          for (char x = 'a'; x <= 'z'; x++)
          {
             table[1, x] = 2;  // (q0,lettre) --> q1
             table[2, x] = 2;  // (q1,lettre) --> q1
          }
          for (char x = '0'; x <= '9'; x++)
            table[2, x] = 2;  // (q1,chiffre) --> q1
          table[2, sentinelle] = EtatFinal;
   }
}
```

Programme utilisant la classe AutomateIdentif

Reconnaissant le langage des identificateurs

```
class ProgramAEFD
{
  static void Main(string[] args)
  {
     AutomateIdentif AEFD_Ident = new AutomateIdentif(3);
     AEFD_Ident.Mot = "v14bcd73";
     AEFD_Ident.Analyser();
  }
}
```

*Exécution sur l'identificateur **prix2** :*

```
(p,1)--> 2
(r,2)--> 2
(i,2)--> 2
(x,2)--> 2
(2,2)--> 2
(#,2)--> 3
Chaîne reconnue.
```

*Exécution sur l'exemple **v14bcd73** :*

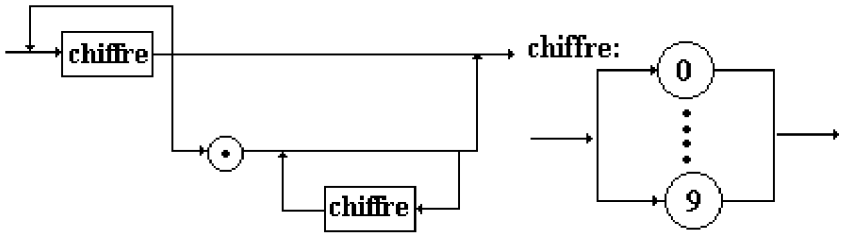
```
(v,1)--> 2
(1,2)--> 2
(4,2)--> 2
(b,2)--> 2
(c,2)--> 2
(d,2)--> 2
(7,2)--> 2
(3,2)--> 2
(#,2)--> 3
Chaîne reconnue.
```

3.4 Exemple : les constantes numériques

On donne des diagrammes syntaxiques de construction des constantes décimales positives comme on peut en trouver dans C# :

1.23
145
589.02
…

constante :



Construisons un programme C# reconnaissant de telles constantes en utilisant le procédé de génération manuelle.

Méthode de travail adoptée :*(identique à la précédente)*

- **Détermination d'une grammaire G adéquate.**
- **Construction de l'automate AEFD associé à G.**
- **Programme C# associé à l'automate construit.**

Détermination d'une grammaire G_{cte} adéquate

Reconnaissons d'abord qu'il y a une grammaire de type 3 engendrant ces identificateurs.

Soient les ensembles :
EnsChiffre = { 0, 1,..., 9 }. *// les 10 chiffres*
V_T = EnsChiffre
V_N = { ⟨*constante*⟩, A, B }
Posons G_{cte} = (V_T , V_N , ⟨ *constante* ⟩ , Règles) une grammaire où
Axiome : ⟨*constante*⟩
Règles :
⟨*constante*⟩ → **0** |**1**|...|**9** A
A → **0** |**1**|...|**9** A
A → ε
A → B
B → **0** |**1**|...|**9** B
B → ε
La grammaire G_{cte} est de type 3 déterministe.

Construction de l'automate associé à G_{cte}

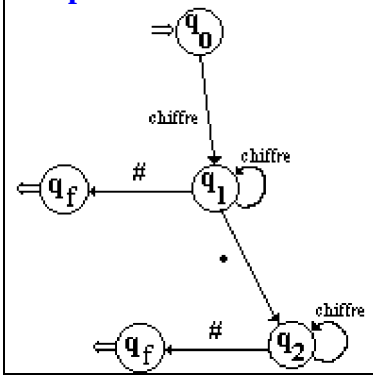
Afin de réduire le nombre de lignes de texte, nous adoptons les mêmes conventions d'écriture que celles de l'exemple précédent:

(q_k , Chiffre) → q_i , *représente l'ensemble des 10 règles :*
(q_k ,**0**) → q_i
.....
(q_k ,**9**) → q_i

Etats associés aux éléments de V_N:	**Règles de l'automate associé à G_{cte} :**
⟨*constante*⟩ ⇔ q_0	(q_0 , Chiffre) → q_1 *// 10 règles*
⟨ A ⟩ ⇔ q_1	(q_1 , Chiffre) → q_1 *// 10 règles*
⟨ B ⟩ ⇔ q_2	(q_1 ,#) → q_f

Etat initial = q_0 Etat final = q_f **Vocabulaire terminal de l'automate :** $V_T' = V_T \cup \{\#\} = \text{Chiffre} \cup \{\#\}$	(q_1 , .) → q_2 (q_2 , Chiffre) → q_2 *// 10 règles* (q_2 ,#) → q_f Soit un total de 33 règles.

Graphe de l'automate :



Matrice de transitions de l'automate:

	0 9	.	#
q_0	q_1 ..... q_1	■	■
q_1	q_1 ..... q_1	q_2	q_f
q_2	q_2 ..... q_2	■	q_f

Programme associé à l'automate

Nous héritons de la classe AutomateAbstr. Comme dans l'exercice précédent, nous n'avons que la méthode **init_table** à redéfinir

```
public class AutomateConstNum  : AEFDAbstr
{
      //-- constructeur :
      public AutomateConstNum  (int end)
      {
         if (end >= 2 & end <= fin)
            EtatFinal = end;
         else
            EtatFinal = fin;
         FMot = sentinelle.ToString();
         init_table();
      }

    protected override int transition(int q, char car)
     // par matrice de transition
     {
            Console.Write("(" + car + "," + q + ")");
            q = table[q, car];
            Console.WriteLine("--> " + q);
            return q;
     }
```

```
protected override void init_table()
{   //par défaut tout est non reconnu:
     for (int i = non; i < fin; i++)
          for (int j = 0; j < 256; j++)
               table[i, j] = non;
      //Les règles de l'AEFD :
      for (char x = '0'; x <= '9'; x++)
      {
         table[1, x] = 2;  // (q0,chiffre) --> q1
         table[2, x] = 2;  // (q1,chiffre) --> q1
         table[3, x] = 3;  // (q2,chiffre) --> q2
      }
      table[2, '.'] = 3;   // (q1,'.') --> q2  }
      table[2, sentinelle] = EtatFinal;   // (q1,#) -->  qf
      table[3, sentinelle] = EtatFinal;   // (q2,#) -->  qf
  }
}
```

Programme utilisant la classe AutomateConstNum

Reconnaissant le langage des constantes numériques

```
class ProgramAEFD
{
  static void Main(string[] args)
  {
    AutomateConstNum  AEFD_CtNum = new AutomateConstNum  (4);
    AEFD_ CtNum.Mot = "145.78";
    AEFD_ CtNum.Analyser();
  }
 }
```

*Exécution sur l'exemple **145** :*

```
(1,1)--> 2
(4,2)--> 2
(5,2)--> 2
(#,2)--> 4
Chaîne reconnue.
```

*Exécution sur l'exemple **145.78** :*

```
(1,1)--> 2
(4,2)--> 2
(5,2)--> 2
(.,2)--> 3
(7,3)--> 3
(8,3)--> 3
(#,3)--> 4
Chaîne reconnue.
```

Le lecteur aura pu se convaincre de la facilité d'utilisation d'un tel générateur manuel. Il lui est conseillé de réécrire un programme personnel fondé sur ces idées. Le polymorphisme dynamique de méthode a montré son utilité dans ces exemples.

Nous appliquons dans le chapitre suivant cette connaissance toute neuve à un petit projet de construction d'un **interpréteur** pour un langage analysable par grammaire de type 3. Nous verrons comment utiliser le graphe d'un automate dans le but de programmer les décisions d'interprétation. Là aussi, le lecteur pourra aisément adapter la méthodologie à d'autres exemples semblables construits sur des K-grammaires.

14.3 : classe d'interpréteur d'un langage de type 3

Objectif : Construction et utilisation d'une classe d'interpréteur d'un langage sans constantes, généré par une grammaire de type 3.

ENONCE :

On donne la Grammaire **G** suivante :

V_N = { ⟨prog.⟩, ⟨bloc⟩, ⟨égal⟩, ⟨suite1⟩, ⟨suite2⟩, ⟨membre droit⟩, ⟨plus⟩, ⟨suite3⟩ , ⟨moins⟩ , ⟨mult⟩ , ⟨div⟩ }
V_T = { **'a'** , **'b'**,..., **'z'** , **'}'** , **'{'** , **';'** , **'L'**, **'E'** , **'+'** , **'='**, **'*'** , **'-'** , **'/'** }

Axiome : ⟨prog.⟩
Règles :
⟨prog.⟩ → **{** ⟨bloc⟩
⟨bloc⟩ → **}**
⟨bloc⟩ → **a**⟨égal⟩ | **b**⟨égal⟩ | . . . | **z**⟨égal⟩
⟨bloc⟩ → **L**⟨suite1⟩ | **E**⟨suite1⟩
⟨suite1⟩ → **a**⟨suite2⟩ | **b**⟨suite2⟩ | . . . | **z**⟨suite2⟩
⟨suite2⟩ → **;** ⟨bloc⟩
⟨égal⟩ → **=** ⟨membre droit⟩
⟨membre droit⟩ → **a**⟨moins⟩ | **b**⟨moins⟩ | . . . | **z**⟨moins⟩
⟨membre droit⟩ → **a**⟨plus⟩ | **b**⟨plus⟩ | . . . | **z**⟨plus⟩
⟨membre droit⟩ → **a**⟨mult⟩ | **b**⟨mult⟩ | . . . | **z**⟨mult⟩
⟨membre droit⟩ → **a**⟨div⟩ | **b**⟨div⟩ | . . . | **z**⟨div⟩
⟨membre droit⟩ → **a**⟨reste⟩ | **b**⟨reste⟩ | . . . | **z**⟨reste⟩
⟨moins⟩ → **-** ⟨suite3⟩ | **;** ⟨bloc⟩
⟨plus⟩ → **+** ⟨suite3⟩ | **;** ⟨bloc⟩
⟨mult⟩ → ***** ⟨suite3⟩ | **;** ⟨bloc⟩
⟨div⟩ → /⟨suite3⟩ | **;** ⟨bloc⟩
⟨reste⟩ → **%** ⟨suite3⟩ | **;** ⟨bloc⟩
⟨suite3⟩ → **a**⟨suite2⟩ | **b**⟨suite2⟩ | . . . | **z**⟨suite2⟩

Questions :

1°) Construire une classe interpréteur de L(G). On donne la sémantique suivante :
(les spécifications sont fournies en langage algorithmique)

Lx correspond à **lire**(x)
Ex correspond à **ecrire**(x)
x = y correspond à **x** ¬ y
a,b,...,z correspondent à des variables contenant des entiers relatifs
+ correspond à l'opérateur d'addition sur les entiers relatifs.
- correspond à l'opérateur de soustraction sur les entiers relatifs.
* correspond à l'opérateur de multiplication sur les entiers relatifs.
/ correspond à l'opérateur de quotient euclidien sur les entiers relatifs.
% correspond à l'opérateur de reste euclidien sur les entiers relatifs.

On utilisera une *mémoire centrale* dans laquelle se trouveront les variables (dans un tableau) et une autre table contenant les noms des variables et leur *adresse* en mémoire centrale (*table des symboles*).

2°) Construire une IHM de calculatrice programmable fondée sur la classe précédente d'interpréteur du langage L(G), calculatrice dans laquelle l'utilisateur peut entrer des lignes de programmes en L(G) et les exécuter.

UNE SOLUTION AU PROBLEME PROPOSE

Spécifications de base du logiciel

1°) Construction d'un analyseur **du langage L(G).**
2°) Construction d'un interpréteur **du langage L(G).**
3°) Le programme console d'interpréteur.
4°) Construction de 2 classes**.**

Démarche adoptée :
Nous adoptons la méthodologie de développement de l'algorithme d'interprétation évoquée au chapitre précédent (construction d'un analyseur puis de l'interpréteur associé), en l'adaptant au langage L(G). Ensuite nous construirons une classe fondée sur cet algorithme.

1°) Construction d'un analyseur du langage L(G)

Nous remarquons que la grammaire **G** est une grammaire de type 3 déterministe(d'état fini). En appliquant le principe de correspondance entre grammaires de type 3 et automates d'états finis, nous construisons l'AEFD associé qui sera un analyseur du langage engendré par **G**.

correspondance entre les éléments de $\mathbf{V_N}$ et les états de l'automate :
<prog.> ⇔ q_0
<suite2> ⇔ q_3
<plus> ⇔ q_6 *idem pour* <moins>, <mult>, et <reste>
<bloc> ⇔ q_1
<egal> ⇔ q_4
<suite3> ⇔ q_7
<suite1> ⇔ q_2
<membre droit> ⇔ q_5

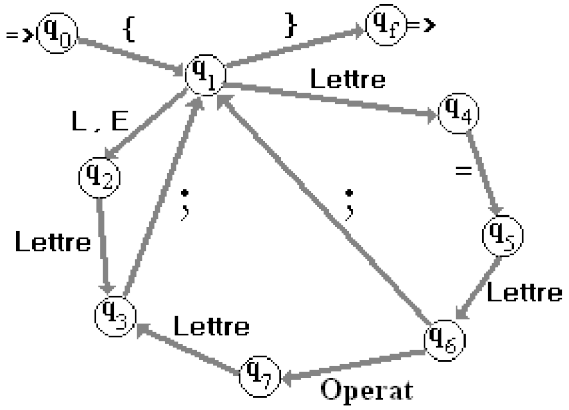
Soit l'AEFD A, $A = (V_T', E, q_0, q_f, \mu)$
$E = \{ \mathbf{q_0}, q_1, q_2, q_3, q_4, q_5, q_6, q_7, \mathbf{q_f} \}$
état initial : $\mathbf{q_0}$
état final : $\mathbf{q_f}$
V_T = { **'a'** , **'b'**,..., **'z'** , **'}'** , **'{'** , **';'** , **'L'**, **'E'** , **'+'** , **'='**, **'*'** , **'-'** , **'/'** }

Nous poserons pour simplifier :
Lettre = {'a', 'b',..., 'z'},
Operat = { '+' , '*' , '-' , '/' , '%' },

Afin de réduire le nombre de lignes de texte nous adoptons la convention d'écriture suivante :

(q_k , **Lettre**) → q_i représente l'ensemble des 26 règles	(q_k ,**a**) → q_i (q_k ,**b**) → q_i ... (q_k ,**z**) → q_i
(q_k , **Operat**) → q_i représente l'ensemble des 5 règles	(q_k ,+) → q_i (q_k , -) → q_i (q_k ,*) → q_i (q_k , /) → q_i (q_k ,%) → q_i

Nous obtenons alors un AEFD dont nous donnons les règles et le graphe.

Règles de transitions de μ	**Graphe de l'automate A**
(q_0, **{**) → q_1 (q_1, **}**) → q_f (q_1, **Lettre**) → q_4 (q_1, **L**) → q_2 (q_1, **E**) → q_2 (q_2, **Lettre**) → q_3 (q_3, **;**) → q_1 (q_4, **=**) → q_5 (q_5, **Lettre**) → q_6 (q_6, **;**) → q_1 (q_6, **Operat**) → q_7 (q_7, **Lettre**) → q_3	

Employons la démarche conseillée dans le cours pour construire la matrice de transition de l'analyseur AEFD, grâce à la méthode init_table de la classe implantant l'automate :

```
public class AutomateMicroLangage: AEFDAbstr
{
      //-- constructeurs :
      public AutomateIdentif (int end)
      {
         if (end >= 2 & end <= fin)
            EtatFinal = end;
         else
            EtatFinal = fin;
         FMot = sentinelle.ToString();
         init_table();
      }
      public AutomateMicroLangage() {       }

    protected override int transition(int q, char car)
     // par matrice de transition
    {
            Console.Write("(" + car + "," + q + ")");
            q = table[q, car];
            Console.WriteLine("--> " + q);
            return q;
    }
```

```
    protected override void init_table()
    {   //par défaut tout est non reconnu:
        for (int i = non; i < fin; i++)
            for (int j = 0; j < 256; j++)
                table[i, j] = non;

        //Les règles de l'AEFD :
        for (char k = 'a'; k <= 'z'; k++)
        {
            table[2, k] = 5;
            table[3, k] = 4;
            table[6, k] = 7;
            table[8, k] = 4;
        }
        table[1, '{'] = 2;          table[2, '}'] = EtatFinal;
        table[2, 'E'] = 3;          table[2, 'L'] = 3;
        table[4, ';'] = 2;          table[5, '='] = 6;
        table[7, '+'] = 8;          table[7, '*'] = 8;
        table[7, '-'] = 8;          table[7, '%'] = 8;
        table[7, '/'] = 8;          table[7, ';'] = 2;
    }
}
```

Rappels des règles de l'automate	Valeurs C# associées
(q0, '{') --> q1 *(q1, '}]') --> qf* *(q1, Lettre) --> q4* *(q1, L) --> q2* *(q1, E) --> q2* *(q2, Lettre) --> q3* *(q3, ;) --> q1* *(q4, =) --> q5* *(q5, Lettre) --> q6* *(q6, ;) --> q1* *(q6, Operat) --> q7* *(q7, Lettre) --> q3*	*Q0 = 1* *Q1 = 2* . . . . *q7 = 8* *qf = 9*

Programme d'analyseur utilisant la classe AutomateMicroLangage

Reconnaissant le langage L(**G**)

```
class ProgramInterprete
{
    static void Main(string[] args)
    {
      AutomateMicroLangage Analang = new AutomateMicroLangage (9);
      Analang.Mot = "{La;Lb;Ea;a=b*c;}";
      Analang.Analyser();
      Console.ReadLine();
    }
 }
```

Exécution sur le mot: **{La;Lb;Ea=b;}**

```
({,1)--> 2
(L,2)--> 3
(a,3)--> 4
(;,4)--> 2
(L,2)--> 3
(b,3)--> 4
(;,4)--> 2
(E,2)--> 3
(a,3)--> 4
(=,4)--> 0
Blocage, chaîne non reconnue.
```

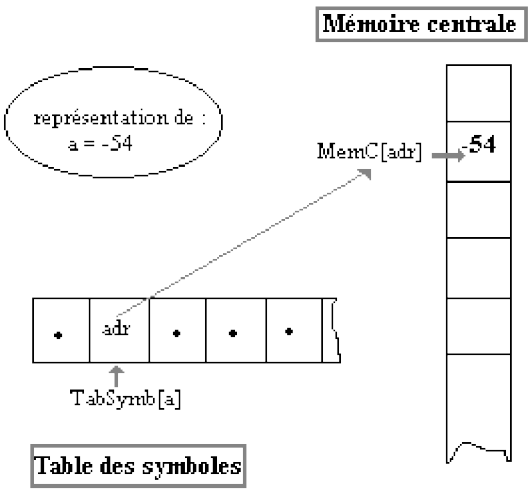
Exécution sur le mot:
{La;Lb;Ea;a=b*c;}

```
({,1)--> 2
(L,2)--> 3
(a,3)--> 4
(;,4)--> 2
(L,2)--> 3
(b,3)--> 4
(;,4)--> 2
(E,2)--> 3
(a,3)--> 4
(;,4)--> 2
(a,2)--> 5
(=,5)--> 6
(b,6)--> 7
(*,7)--> 8
(c,8)--> 4
(;,4)--> 2
(},2)--> 9
Chaîne reconnue.
```

2°) Construction d'un interpréteur à partir de l'AEFD

Rappelons les spécifications proposées par l'énoncé :

Lx correspond à **lire**(x) **E**x correspond à **ecrire**(x) x = y correspond à **x** ← y **a**,**b**,..., **z** correspondent à des variables entiers relatifs	**Le mécanisme d'accès est simple :**

+ correspond à l'opérateur d'addition sur les entiers relatifs. - correspond à l'opérateur de soustraction sur les entiers relatifs. * correspond à l'opérateur de multiplication sur les entiers relatifs. / correspond à l'opérateur de quotient euclidien sur les entiers relatifs. % correspond à l'opérateur de reste euclidien sur les entiers relatifs.	

Spécifications opérationnelles

L'énoncé nous propose une spécification d'implantation en C# pour la mémoire centrale (nous la dénommons **MemC**)et la table des symboles (nous la dénommons **TabSymb**).

Le tableau **TabSymb** est indexé directement sur les caractères (ce qui évite la construction d'une fonction de codage). Chaque cellule **TabSymb**['a'], **TabSymb**['b'],..., **TabSymb**['z'], contient un nombre qui est l'adresse adr (numéro de case dans **MemC**) de la variable a, b,..,z.

A partir de ce numéro de case adr, la cellule MemC[adr] du tableau MemC contient la valeur numérique de la variable d'adresse adr.

Spécifications d'implantation pour les instructions

En partant des spécifications de données précédentes, nous pouvons proposer une implantation de chacune des instruction du langage L(G).

Nous noterons par la suite, ≅ la relation de traduction en pseudo C#.

Nous avons utilisé trois métasymboles x,y et z pour remplacer le nom d'une quelconque des variables **a**, **b**…etc. afin de ne pas alourdir la traduction par de nombreuses lignes redondantes.

Interprétation de l'instruction L :

Lx	≅	TabSymb[x] = adressecourante ; adressecourante = adressecourante +1 ; MemC[TabSymb[x]] = Convert.ToInt32(Console.ReadLine());

Interprétation de l'instruction E :

Ex	≅	adressecourante := TabSymb[x]; Console.WriteLine(MemC[adressecourante]) ;

Interprétation de l'instruction x = y :

`x = y` ≅	MemC[TabSymb[x]] = MemC[TabSymb[y]] ;

*Interprétation de l'instruction x = y **oper** z :*

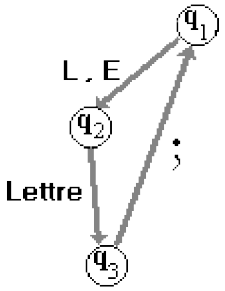
`x=y` **`oper`** `z` ≅ ***oper*** *l'un des opérateurs suivants :* *{ '+' , '*' , '-' , '/, '%' }*	MemC[TabSymb[x]] = MemC[TabSymb[y]] **oper** MemC[TabSymb[z]]

Nous allons construire notre interpréteur à partir du graphe de l'AEFD que nous possédons.

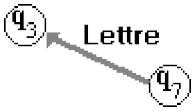
Ainsi, nous passerons de la version analyseur à la version interpréteur en suivant les chemins du graphe et en repérant les points clefs où l'interprétation d'une instruction est possible. Nous adoptons comme stratégie le fait que le lancement d'une interprétation ne sera possible que lorsque l'on sera arrivé dans le graphe à un endroit où une instruction vient juste d'être reconnue.

Interprétation des instructions Ex et Lx

Nous observons tout d'abord sur la portion de graphe de l'AEFD comment les instructions Ex et Lx sont reconnues.



Une telle instruction est entièrement reconnue lorsque l'automate est à l'état q_3. On pourrait lancer l'interprétation de Ex ou Lx, mais ce serait une erreur car il y a un autre chemin dans le graphe qui amène à l'état q_3.



Nous voyons qu'il n'y a que deux manières différentes d'arriver en q_3 : soit en venant de q_2, soit en venant de q_7.

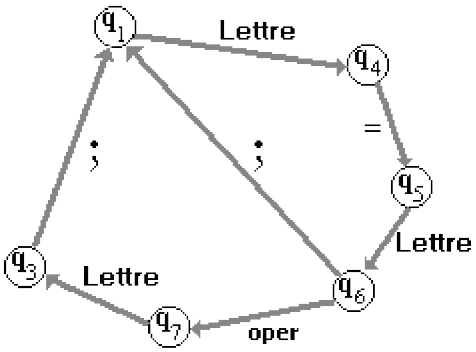
> Il nous faut une variable que nous nommerons **etatavant** qui mémorisera l'état précédent. Le symbole qui vient d'être analysé est stocké dans une variable que nous nommons **symlu**.
> D'autre part, lorsque nous sommes en q_3, le symbole qui vient d'être analysé est un élément de {a,b,c}. Afin de décider s'il s'agit d'une instruction Ex ou bien Lx, il nous faut avoir mémorisé le symbole précédent qui a été analysé en passant de q_1 à q_2. Nous nommerons cette variable **symprec**.

Voici donc en pseudo-C# le code de lancement de l'interprétation de Lx ou de Ex.	**if** (etat==q_3 & etatavant==q_2 & symprec==L) **{** *// on interprète le Lx* TabSymb[symlu] = adressecourante ; adressecourante ++ ; Lire(symLu, out MemC[TabSymb[symLu]]); **}**

Ce code est à rajouter à l'AEFD précédent.	**else** *// on interprète le Ex* **{** adressecourante = TabSymb[symlu]; Ecrire(symLu, MemC[adresseCourante]); **}**

Interprétation de l'instruction x = y oper z

Nous avons vu que l'autre façon d'arriver à l'état q_3 est d'avoir suivi l'arc q_7 à q_3.



Ce chemin correspond très exactement à l'analyse d'une instruction x = y **oper** z. Afin de pouvoir interpréter correctement cette instruction, nous devons avoir mémorisé les noms des variables x, y et z le long du chemin d'analyse : q_1 à q_4 à q_5 à q_6 à q_7 à q_3. Afin de ne pas rajouter trop de nouveaux états nous avons décidé de n'avoir qu'une transition sur un ensemble d'opérateurs (q_6 **oper**) à q_7. Nous regroupons alors les symboles +,-,*,/,% dans un même ensemble (**Operat** : set of Vt), que nous initialiserons dans la procédure d'initialisation : **Operat =[**'+', '-', '*', '/', '%' **]**

Nous notons que :
x est connu lorsque l'AEFD est à l'état q_4 (à la fin de l'arc q_1 à q_4)
y est connu lorsque l'AEFD est à l'état q_6 (à la fin de l'arc q_5 à q_6)
z est connu lorsque l'AEFD est à l'état q_3 (à la fin de l'arc q_7 à q_3)

Le stockage d'un troisième symbole de variable nécessite l'utilisation d'une nouvelle variable servant à le mémoriser. Nous la nommerons **symavant**.

En résumant la situation, nous aurons pour x = y **oper** z :

- z est stocké dans **symlu**, il est reconnu à l'état q_3.
- y est stocké dans **symprec**, il est reconnu à l'état q_6.
- x est stocké dans **symavant**, il est reconnu à l'état q_4.
- l'opérateur **oper** est reconnu à l'état q_7, il est alors rangé dans une variable **OperVar** de type Vt servant à cet effet.

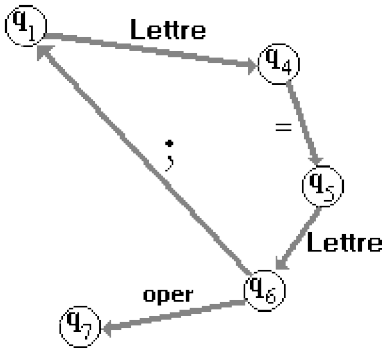
Voici donc en pseudo-C# le code de lancement de l'interprétation de l'instruction x = y **oper** z. Ce code est aussi à rajouter au code précédent.

```
if ( etat==q4 ) symavant = symlu ;
if ( etat=q6 ) symprec = symlu ;
if ( etat=q7 )
  if ( symlu Î Operat ) OperVar = symlu;
if (etat==q3 & etatavant==q7) //on interprète x = y oper z}
  switch ( OperVar ) {
    case '+ ': MemC[TabSymb[symavant]]=MemC[TabSymb[symprec]] + MemC[TabSymb[symlu]];
```

```
  case '-': MemC[TabSymb[symavant]]=MemC[TabSymb[symprec]] - MemC[TabSymb[symlu]];
  case '*': MemC[TabSymb[symavant]]=MemC[TabSymb[symprec]] * MemC[TabSymb[symlu]];
  case '/': MemC[TabSymb[symavant]]=MemC[TabSymb[symprec]] /  MemC[TabSymb[symlu]];
  case '%': MemC[TabSymb[symavant]]=MemC[TabSymb[symprec]] % MemC[TabSymb[symlu]];
}
```

Interprétation de l'instruction x=y

Si nous observons la partie du graphe de l'AEFD correspondant à l'analyse de l'instruction x=y, nous remarquons que contrairement aux cas précédents ce n'est pas à l'état q_6 que nous pouvons prendre une décision.



En effet, à partir de q_6 il y a deux possibilités d'instructions : soit x=y, soit x = y **oper** z. Il nous faut aller un état plus loin dans le graphe (déterministe) afin de décider si l'on est dans un cas ou dans l'autre.

Si l'état d'après q_6 est q_1, il s'agit d'une instruction x=y, si l'état d'après q_6 est q_7 il s'agit d'une instruction x=y+z.

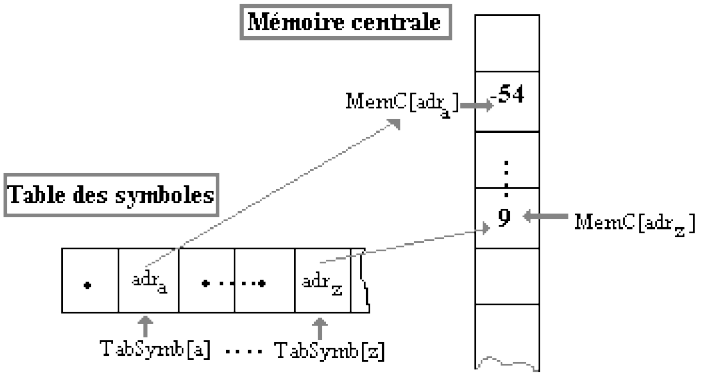
Le chemin d'analyse d'une instruction x=y est :
q_1 à q_4 à q_5 à q_6 à q_1.
Comme dans l'analyse du problème précédent nous avons :

- **;** est stocké dans **symlu**, il est reconnu à l'état q_1.
- y est stocké dans **symprec**, il est reconnu à l'état q_6.
- x est stocké dans **symavant**, il est reconnu à l'état q_4.

Voici donc en pseudo-C#, le code de lancement de l'interprétation de l'instruction x=y. Ce code est le dernier à rajouter au code précédent.

```
If (etat==q1 & etatavant==q6) //on interprète x = y
   MemC[TabSymb[symavant]] = MemC[TabSymb[symprec]]
```

Le mécanisme d'accès implanté
Par TabSymb et MemC :



Le lecteur pourra étendre le programme en augmentant par exemple le nombre de variables, ou le nombre d'opérateurs.

Voici ci-dessous les squelettes des méthodes C# d'une classe abstraite d'interpréteur InterpreteurAbstr héritant AutomateMicroLangage permettant d'étendre l'analyseur en interpréteur abstrait :

```
private void ClearMemC(){…}
 // RAZ mémoire centrale

private void ClearTabSymb(){…}
 // RAZ table des symboles

 private void Initialisations(){…}
 // RAZ tout

private void Exec(string chaine, out int etat) {…}
// moteur d'analyse et d'interpretation de l'automate}

private bool isInTabSymb(char identif) {…}
  // indique si le symbole identif est deja dans TabSymb}

private bool isInOperateur(char symb) ) {…}
// indique si le symbole symb appartient à l'ensemble des opérateurs}
```

Comme notre interpréteur doit lire et analyser un texte de programme et écrire les resultats d'exécution, nous proposons d'écrire une classe qui contienne toutes spécifications nécessaires et utiles au fonctionnement d'un interpréteur, mais qui ne dépende pas de la façon dont on lit et dont on affiche les résultats. Pour cela nous avons proposé de construire une classe abstraite InterpreteurAbstr qui possède 2 méthodes abstraites (donc non implémentées) **Lire** et **Ecrire**.

Le soin d'expliquer comment lire ou écrire est délégué à une classe **'concrète'** qui dérivera de InterpreteurAbstr.

1°) Construction d'une classe abstraite d'interpréteur de L(G)

Nous ajoutons à cette classe, en visibilité **protected,** les deux méthodes abstraites :

```
protected abstract void Lire(char symb, out int x); // symb = la variable à lire, x = sa valeur entière
protected abstract void Ecrire(char symb, int x); // symb = la variable à ecrire, x = sa valeur entière
```

Ci-après une implémentation de la classe abstraite InterpreteurAbstr :

```
abstract class InterpreteurAbstr : AutomateMicroLangage
{
   //etats : q0=1,q1=2,q2=3,q3=4,q4=5,q5=6,q6=7,q7=8,qf=9
   private const int maxAdresse = 1024;
   private int[] MemC = new int[maxAdresse];
   private int[] TabSymb = new int[256];
   private char OperVar;
   private int numcar;
   private void ClearTabSymb()
   {
      for (int i = 0; i < 256; i++)
         TabSymb[i] = 0;
   }
   private void ClearMemC()
   {
      for (int i = 0; i < maxAdresse - 1; i++)
         MemC[i] = 0;
   }
   private void Initialisations()
   {
      ClearMemC();
      ClearTabSymb();
   }
   private bool isInTabSymb(char identif)
   {
      return TabSymb[identif] != 0;
   }
   private bool isInOperateur(char symb)
   {
      return symb == '+' | symb == '-' | symb == '*' | symb == '/' | symb == '%';
   }
   private void Exec(string chaine, out int etat)
   {
      char symLu;
      int adresseCourante = 1;
      char symPrec, symAvant;
      int etatAvant;
      numcar = 0;
      etat = 1;
      Mot = chaine;
      symPrec = (char)0;
      symAvant = (char)0;
      while (etat != non & etat != EtatFinal)
      {
         symLu = Symsuiv(numcar++);
         etatAvant = etat;
         etat = transition(etat, symLu);
         if (etat == 3) symPrec = symLu;
         if (etat == 5)
         {
            symAvant = symLu;
```

```
            if (!isInTabSymb(symLu))
            {
               TabSymb[symLu] = adresseCourante;
               adresseCourante++;
            }
         }
         if (etat == 7) symPrec = symLu;
         if (etat == 8)
            if (isInOperateur(symLu))
               OperVar = symLu;
         if (etat == 4 & etatAvant == 3 & symPrec == 'L')// Lx;
         {
            TabSymb[symLu] = adresseCourante;
            adresseCourante++;
            Lire(symLu, out MemC[TabSymb[symLu]]);
         }
         else
            if (etat == 4 & etatAvant == 3 & symPrec == 'E')// Ex;
            {
               adresseCourante = TabSymb[symLu];
               Ecrire(symLu, MemC[adresseCourante]);
            }
            else
               if (etat == 2 & etatAvant == 7)// x=y;
                  MemC[TabSymb[symAvant]] = MemC[TabSymb[symPrec]];
               else
                  if (etat == 4 & etatAvant == 8)// x = y oper z;
                     switch (OperVar)
                     {
                        case '+': MemC[TabSymb[symAvant]] =
                                    MemC[TabSymb[symPrec]] + MemC[TabSymb[symLu]]; break;
                        case '-': MemC[TabSymb[symAvant]] =
                                    MemC[TabSymb[symPrec]] - MemC[TabSymb[symLu]]; break;
                        case '*': MemC[TabSymb[symAvant]] =
                                    MemC[TabSymb[symPrec]] * MemC[TabSymb[symLu]]; break;
                        case '/': MemC[TabSymb[symAvant]] =
                                    MemC[TabSymb[symPrec]] / MemC[TabSymb[symLu]]; break;
                        case '%': MemC[TabSymb[symAvant]] =
                                    MemC[TabSymb[symPrec]] % MemC[TabSymb[symLu]]; break;
                     }
   }
}
protected abstract void Lire(char symb, out int x);
protected abstract void Ecrire(char symb, int x);

public string Filtrage(string Texte)
{
   string s = "";
   for (int i = 0; i < Texte.Length; i++)
      if (Texte[i] >= 'a' & Texte[i] <= 'z' |
         Texte[i] == ';' |
         Texte[i] == '{' |
         Texte[i] == '}' |
         Texte[i] == 'E' |
         Texte[i] == 'L' |
         Texte[i] == '+' |
         Texte[i] == '-' |
         Texte[i] == '/' |
         Texte[i] == '*' |
         Texte[i] == '%' |
```

```
            Texte[i] == '='
            )
            s += Texte[i];
        return s;
    }
    public void Lancer(string prog)
    {
        int q;
        Exec(prog, out q);
        if (q == non)
        {
            Console.WriteLine("Blocage erreur de syntaxe :");
            Console.WriteLine(prog.Substring(1,numcar)+"<<-- Erreur.");
        }
        else
            Console.WriteLine("Exécution terminée.");
    }
    public InterpreteurAbstr() : base()       {
    }
    public InterpreteurAbstr (int end): base(end)
    {
        Initialisations();
    }
}
```

Nous avons rajouté une méthode de filtrage du texte source permettant une saisie plus libre du texte (avec des blancs, des sauts de ligne,...) et épurant le texte entré de tous ces éléments :

```
public string Filtrage(string Texte)
```

Le texte suivant entré au clavier sur 8 lignes, soumis à la méthode de filtrage est restitué pour analyse une fois épuré :

{
Lb; Lc;
a = c + b ;
d=a*c;
Ea; Eb;
Ec;
Ed;
}#
	{Lb;Lc;a=c+b;d=a*c;Ea;Eb;Ec;Ed;}#

La méthode « public void Lancer(string prog) » sert à appeler la méthode privé « Exec » (moteur d'analyse de l'automate couplé à l'interprétation).

Afin de faire fonctionner effectivement notre interpréteur, il suffira de construire une classe concrète héritant de la classe abstraite qui redéfinira les méthode Lire et Ecrire et qui lancera le processus d'analyse et d'interprétation d'un « programme » écrit en L(G).

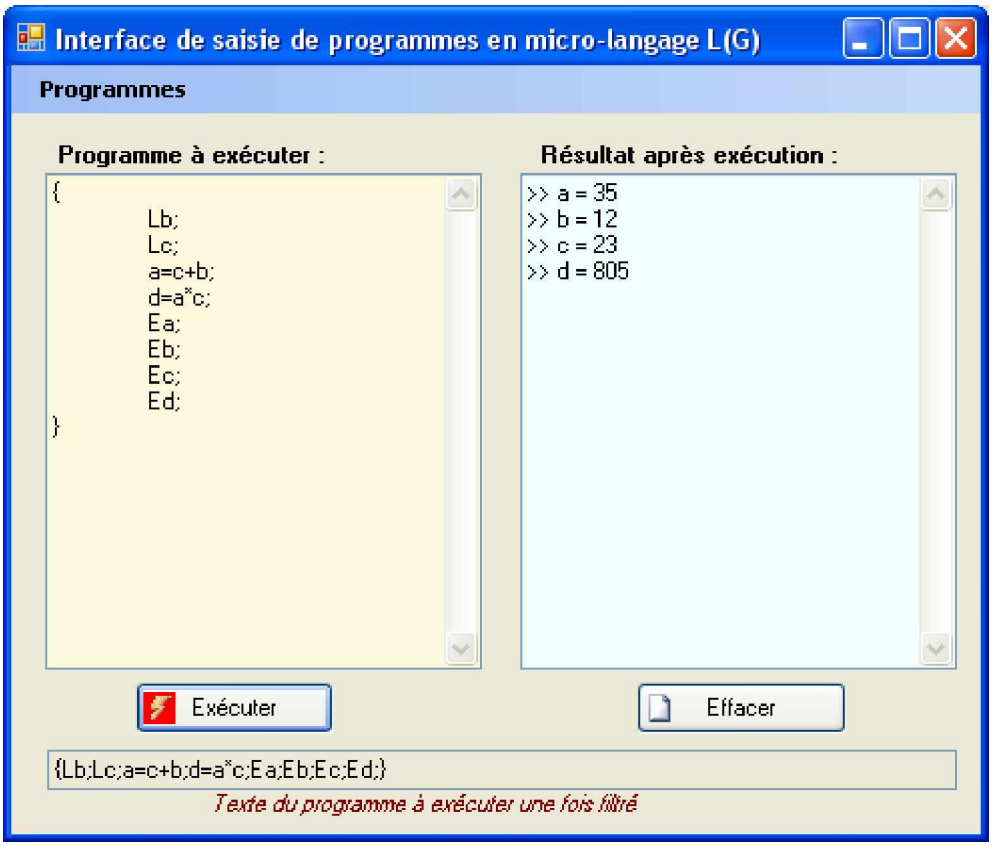
2°) Construction de deux classes héritant de InterpreteurAbstr

2.1°) **Application console - première classe concrète dérivée**

Nous donnons ci-dessous le code d'une classe « concrète » InterpreteurConsole implantant la classe abstraite InterpreteurAbstr correspondant à **une application console** :

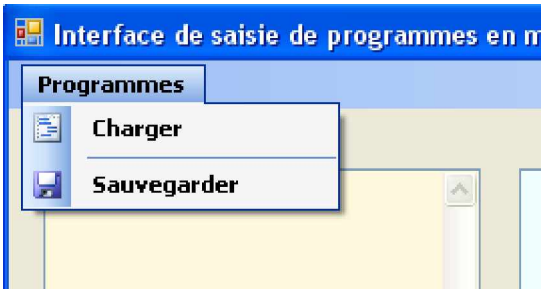
```
class InterpreteurConsole : InterpreteurAbstr
{
   public InterpreteurConsole(int end) : base(end)
   {
   }
   public InterpreteurConsole() : base()
   {
   }
   protected override void Lire(char symb, out int x)
   {
       Console.Write("Entrez : "+symb+" : ");
       x = Convert.ToInt32(Console.ReadLine());
   }
   protected override void Ecrire(char symb, int x)
   {
      Console.WriteLine(">> " + symb + " = " + x);
   }
}
```

2.2°) **Application IHM - deuxième classe concrète dérivée**

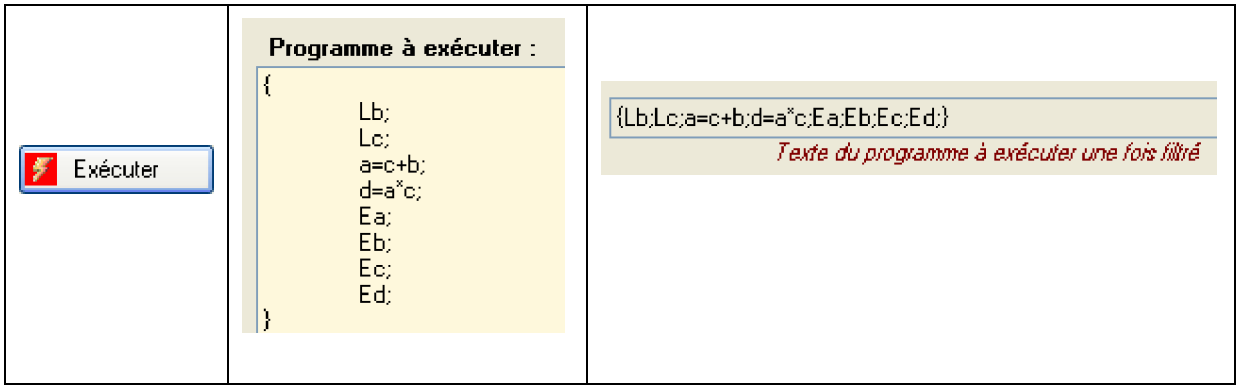


Nous proposons au lecteur d'écrire lui-même **une application IHM** utilisant la classe d'interpréteur InterpreteurAbstr selon le modèle d'interface précédente.

Le programme à analyser et exécuter est entré dans un « TextBox » soit manuellement, soit par chargement d'un fichier de texte grâce à un menu :

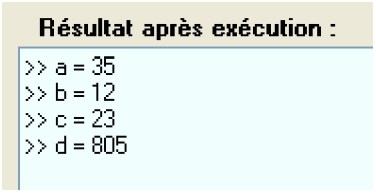


- Le bouton « Exécuter », appelle la méthode de filtrage sur le texte du programme présent dans le « TextBox » et le range dans un autre « TextBox » :



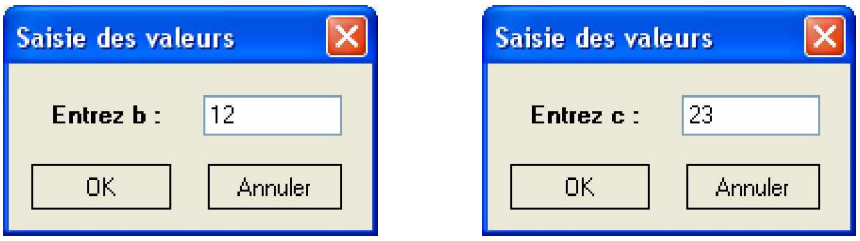
- Le bouton « Exécuter » appelle ensuite, la méthode « **Lancer(...)** » de la classe InterpreteurAbstr .

- Les résultats seront affichés par une redéfinition de la méthode « **Ecrire(...)** » à travers un TextBox.



- Le bouton « Effacer » supprime les affichages du TextBox des résultats :

- Les valeurs seront saisies par une redéfinition de la méthode « **Lire(…)** » à travers un **TextBox** dans une autre fiche :



Voici à titre indicatif, les classes partielles construites avec visual C#

```
// Evénements sur la fiche d'IHM  :
public partial class FInterprete : Form
   {
      private InterpreteurIHM interpr;
      internal FEntree formSaisie;
      public FInterprete()
      {
         InitializeComponent();
         interpr = new InterpreteurIHM(9,this);
      }

      private void buttonExec_Click(object sender, EventArgs e)
      {
         if (textBoxSaisie.Text != "")
         {
            textBoxProg.Text = interpr.Filtrage(textBoxSaisie.Text);
            interpr.Lancer(textBoxProg.Text);
         }
      }

      private void chargerToolStripMenuItem_Click(object sender, EventArgs e)
      {
         if (openFileDialog1.ShowDialog() == DialogResult.OK)
            textBoxSaisie.Text = File.ReadAllText(openFileDialog1.FileName);
      }

      private void sauvegarderToolStripMenuItem_Click(object sender, EventArgs e)
      {
         if (saveFileDialog1.ShowDialog() == DialogResult.OK)
            File.WriteAllText(saveFileDialog1.FileName, textBoxSaisie.Text);
      }

      private void buttonEffacer_Click(object sender, EventArgs e)
      {
         textBoxResult.Clear();
      }
   }
```

```
// classe d'interpréteur concrète :
class InterpreteurIHM : InterpreteurAbstr
 {
    private FInterprete ficheIhm;
    public InterpreteurIHM(int end, FInterprete formulaire ) : base(end)
    {
       ficheIhm = formulaire;
    }
    public InterpreteurIHM(): base()
    {
    }
    protected override void Lire(char symb, out int x)
    {
       ficheIhm.formSaisie = new FEntree();
       ficheIhm.formSaisie.labelEntree.Text = "Entrez " + symb + " : ";
       ficheIhm.formSaisie.ShowDialog(ficheIhm);
       x = Convert.ToInt32(ficheIhm.formSaisie.textBoxEntree.Text);
    }
    protected override void Ecrire(char symb, int x)
    {
       ficheIhm.textBoxResult.AppendText(">> " + symb + " = " + x + "\r\n");
    }
 }

// fiche de saisie des valeurs :
 partial class FEntree : Form
 {
    public FEntree()
    {
       InitializeComponent();
    }

    private void ButtonOk_Click(object sender, EventArgs e)
    {
       this.Close();
    }

    private void ButtonAnnuler_Click(object sender, EventArgs e)
    {
       textBoxEntree.Text = "0";
    }
 }
```