Chapitre 6 : Utiliser des grammaires pour programmer

6.1.Programmation avec les grammaires de type 2

- programmation par les grammaires
- C-grammaire et automate à pile de mémoire

6.2.Automates et grammaires de type 3

- automates pour les grammaires de type 3
- implantation d'un automate d'état fini en pascal
  - déterministe à partir des règles
  - déterministe à partir de sa table des transitions
  - non-déterministe à partir des règles
  - automate pour les identificateurs
  - automate pour les constantes numériques

6.3.Projet d'interpréteur de micro-langage

- la grammaire du micro-langage
- construction de l'analyseur
- construction de l'interpréteur

6.4.Projet d'indentateur de code Delphi

- spécifications
- architecture
- implantation

6.1 Programmation avec des C-grammaires

Plan du chapitre:

1. Programmation par les grammaires (programme en Delphi et Java)

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

$G = (V_N, V_T, S, R)$ → 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 parallèle en Delphi et en Java.

Exemple fictif :

grammaire	*Traduction en Delphi*	*Traduction en Java*
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 Java*

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 Java*

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 Delphi 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 Java*

$\mathbf{V_N}$ = {⟨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 $\mathbf{V_T}$

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 Delphi :

Ces cinq ensembles sont donc représentés en Delphi chacun par un tableau de chaînes. Nous définissons le type **mot** comme le type général, puis cinq tableaux de type mot.

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

```
type
   mot = array[1..Maxnbmot]of string;
champs
   tnom, tadjectif, tarticle ,tverbe, tadverbe : mot;
```

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 à ce niveau de construction :

```
const
   Maxnbmot=4; // nombre maximal de mots dans un tableau
type
   mot = array[1..Maxnbmot]of string;

Gener_fr = class
  private
    tnom, tadjectif, tarticle, tverbe, tadverbe : mot;
    procedure GN;
    procedure GV;
    procedure Art;
    procedure Nom;
    procedure Adj;
    procedure Adv;
    procedure verbe;
 public
    constructor Creer; // constructeur d'objet
    procedure phrase; // axiome de la grammaire
end;
```

Spécification d'implantation en Java : Les spécifications sont les mêmes qu'en Delphi

Etat de la classe Java à ce niveau de construction :

```
   class  Gener_fr {
     final 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 Delphi associées :

```
procedure Gener_fr.initnom;
begin
 tnom[1]:='chat';
 tnom[2]:='chien';
end;
```

```
procedure Gener_fr.initverbe;
begin
 tverbe[1]:='poursuit';
 tverbe[2]:='aime';
end;
```

```
procedure Gener_fr.initadjectif;
begin
 tadjectif[1]:='beau';
 tadjectif[2]:='gentil';
 tadjectif[3]:='noir';
 tadjectif[4]:='blanc';
end;
```

```
procedure Gener_fr.initarticle;
begin
 tarticle[1]:='le';
 tarticle[2]:='un';
end;
```

```
procedure Gener_fr.initadverbe;
begin
 tadverbe[1]:='malicieusement';
 tadverbe[2]:='joyeusement';
end;
```

Ces cinq méthodes sont appelées dans une méthode générale d'initialisation du vocabulaire V_T tout entier.

```
procedure Gener_fr.initabl;
begin
   initnom;
   initarticle;
   initverbe;
   initadjectif
end;
```

Initialisation des contenus en Java :

```
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 ( );
}
```

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:

Delphi	Java

function Gener_fr.Alea(n:integer):integer;
begin
  result := trunc(random*100)mod n+1;
end;
	private Random ObjAlea = **new** Random();
	
	int Alea (**int** n) **{**
	  return ObjAlea.nextInt(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.

Delphi	Java

procedure Gener_fr.phrase;
begin
  GN;
  GV;
  GN;
  writeln('.')
end;
	void phrase ( )
	{
	  GN ( );
	  GV ( );
	  GN ( );
	  System.out.println('.') ;
	}

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.

Delphi	Java

procedure Gener_fr.GN;
begin
if Alea(2)=1 **then** *// pour règle 3*
begin
Art;
Nom;
Adj
end
else *// pour règle 2*
begin
Art;
Adj;
Nom
end
end ;
	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* ⟩

Delphi	Java
procedure Gener_fr.GV; **begin** **if** Alea(2)=1 **then** *// règle: < verbe >* Verbe **else** *// règl:e < verbe >< Adv >.* **begin** Verbe; Adv **end** **end**;	**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 writeln(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
```

Delphi	Java
procedure Gener_fr.Art; **begin** write(tarticle[Alea(2)],' ') **end**;	**void** Art ( ) **{** System.out.print(tarticle[Alea(2)]+' ') ; **}**
procedure Gener_fr.Adj; **begin** write(tadjectif[Alea(4)],' ') **end**;	**void** Adj ( ) **{** System.out.print(tadjectif [Alea(4)]+' ') ; **}**
procedure Gener_fr.Verbe; **begin** write(tverbe[Alea(2)],' ') **end**;	**void** Verbe ( ) **{** System.out.print(tverbe [Alea(2)]+' ') ; **}**
procedure Gener_fr.Nom; **begin** write(tnom[Alea(2)],' ') **end**;	**void** Nom ( ) **{** System.out.print(tnom [Alea(2)]+' ') ; **}**
procedure Gener_fr.Adv; **begin** write(tadverbe[Alea(2)],' ') **end**;	**void** Adv ( ) **{** System.out.print(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 Delphi 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 Delphi	Classe Java
unit UclassGenerFr; **interface** **const** Maxnbmot=4; **type** mot=array[1..Maxnbmot]**of** string; Gener_fr = **class** **private** tnom,tadjectif,tarticle,tverbe,tadverbe: mot; **procedure** initnom; **procedure** initverbe; **procedure** initadverbe; **procedure** initadjectif; **procedure** initarticle; **procedure** initabl; **function** Alea(n:integer):integer; **procedure** Article; **procedure** Nom; **procedure** Adjectif; **procedure** Adverbe; **procedure** Verbe; **procedure** fin; **procedure** Grp_Nom; **procedure** Grp_Verbal; **public** **constructor** Creer;	**import** java.util.Random; **class** Gener_fr **{** **final** int Maxnbmot=4; **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(); **public** Gener_fr( ) **{** initabl ( ); **}** **private void** initnom ( ) **{** tnom[0] ="chat"; tnom[1] = "chien"; **}** **private void** initadjectif ( ) **{** tadjectif[0] = "beau"; tadjectif[1] = "gentil"; tadjectif[2] = "noir"; tadjectif[3] = "blanc";

```
 procedure phrase;
end;

implementation

procedure Gener_fr.initnom; begin
 tnom[1]:='chat';
 tnom[2]:='chien';
end;

procedure Gener_fr.initverbe; begin
 tverbe[1]:='poursuit';
 tverbe[2]:='aime';
end;

procedure Gener_fr.initadverbe; begin
 tadverbe[1]:='malicieusement';
 tadverbe[2]:='joyeusement';
end;

procedure Gener_fr.initadjectif; begin
 tadjectif[1]:='beau';
 tadjectif[2]:='gentil';
 tadjectif[3]:='noir';
 tadjectif[4]:='blanc';
end;

procedure Gener_fr.initarticle; begin
 tarticle[1]:='le';
 tarticle[2]:='un';
end;

procedure Gener_fr.initabl; begin
 Randomize;
 initnom;
 initarticle;
 initverbe;
 initadverbe;
 initadjectif
end;

function Gener_fr.Alea(n:integer):integer;
begin
 Alea:=random(n)+1;
end;

procedure Gener_fr.Article; begin
 write(tarticle[Alea(2)],' ')
end;

procedure Gener_fr.Nom; begin
 write(tnom[Alea(2)],' ')
end;

procedure Gener_fr.Adjectif; begin
 write(tadjectif[Alea(4)],' ')
end;

procedure Gener_fr.Adverbe; begin
 write(tadverbe[Alea(2)],' ')
```

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

private void initverbe ( )
{
 tverbe[0] = "poursuit";
 tverbe[1] = "aime";
}
private void initarticle ( )
{
 tarticle[0] = "le";
 tarticle[1] = "un";
}
private void initabl ( )
{
 initnom ( );
 initarticle ( );
 initverbe ( );
 initadjectif ( );
 initadverbe ( );
}
int Alea(int n)
{
 return ObjAlea .nextInt(n);
}
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 // règle: < verbe > < Adv >
 {
  Verbe ( );
  Adv ( );
 }

}
private void Art  ( )
{
 System.out.print(tarticle[Alea(2)]+' '   ) ;
}
```

```
end;

procedure Gener_fr.Verbe; begin
 write(tverbe[Alea(2)],' ')
end;

procedure Gener_fr.fin; begin
 writeln('.')
end;

procedure Gener_fr.Grp_Nom; begin
 if Alea(2)=1 then
 begin
  Article;
  Nom;
  Adjectif
 end
 else
 begin
  Article;
  Adjectif;
  Nom
 end

end;
procedure Gener_fr.Grp_Verbal; begin
 if Alea(2)=1 then
  Verbe
 else
 begin
  Verbe;
  Adverbe
 end
end;

procedure Gener_fr.phrase; begin
 Grp_Nom;
 Grp_Verbal;
 Grp_Nom;
 fin
end;

constructor Gener_fr.Creer; begin
 initabl;
end;

end.
```

```
 private void Nom  ( )
 {
  System.out.print(tnom[Alea(2)]+' '  ) ;
 }

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

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

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

 private void fin ( )
 {
  System.out.println('.') ;
 }

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

Utiliser la classe Delphi	Utiliser la classe Java

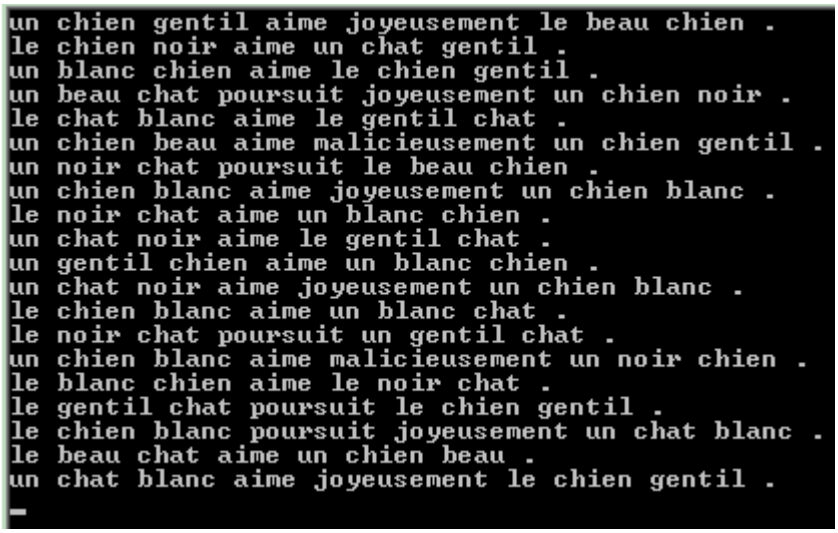
```
program ProjGenerFr;

 {$APPTYPE CONSOLE}

uses
 SysUtils,
 UclassGenerFr in 'UclassGenerFr.pas';
var
 miniFr:Gener_fr;
 i:integer;
```

```
public class UtiliseGenerFr
{
 public static void main(String[] args)
 {
   Gener_fr miniFr = new Gener_fr();
   for(int i=0;i<10;i++)
     miniFr.phrase( );
  }
 }
```

```
begin
 miniFr:=Gener_fr.Creer;
 for i:=1 to 20 do
  miniFr.phrase;
 readln
end.
```

Delphi

Java



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 $V_p \rightarrow$ 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 :

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

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.

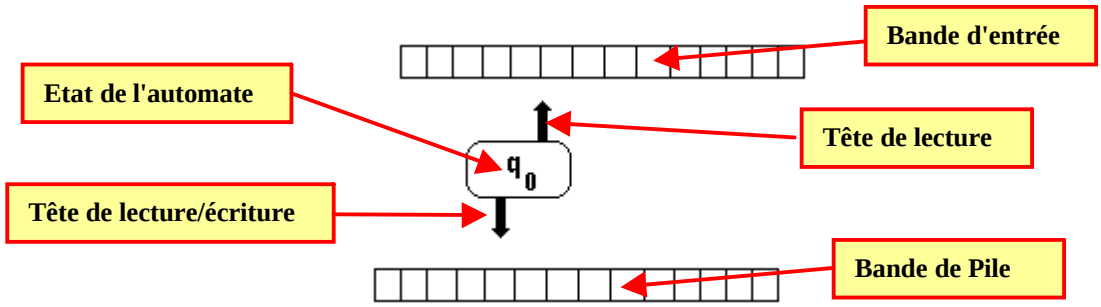
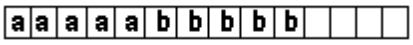


fig - un automate à pile

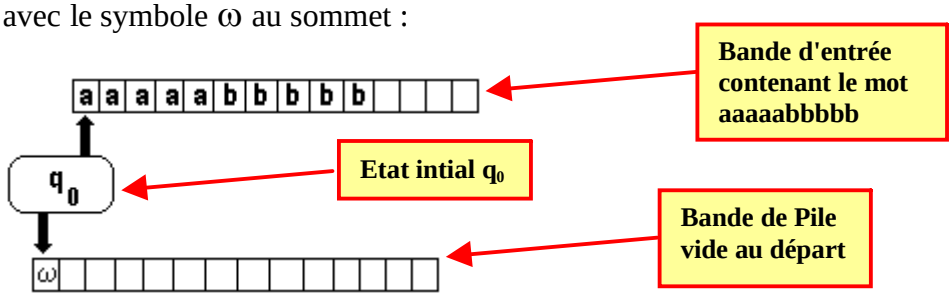
- Les règles de transitions spécifient la manipulation de la bande d'entrée et de la pile de l'automate.

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

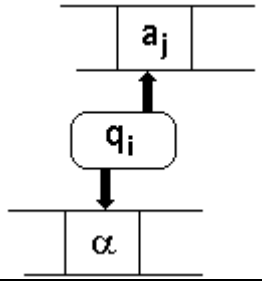
On fournit un mot que l'on écrit symbole par symbole de gauche à droite dans chaque case de l'automate (par exemple avec $\mathbf{V_T}$ ={a,b} le mot **aaaaabbbbb**):



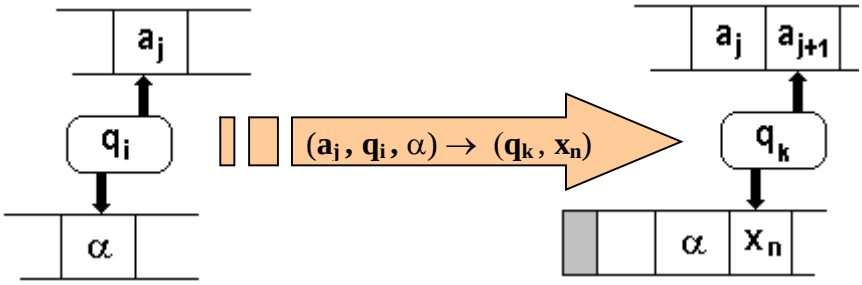
L'automate est mis à l'état initial $\mathbf{q_0}$, sa tête de lecture d'entrée est positionnée sur la première case à gauche de la bande d'entrée(1er symbole du mot à reconnaître), la pile est initialisée avec le symbole ω au sommet :



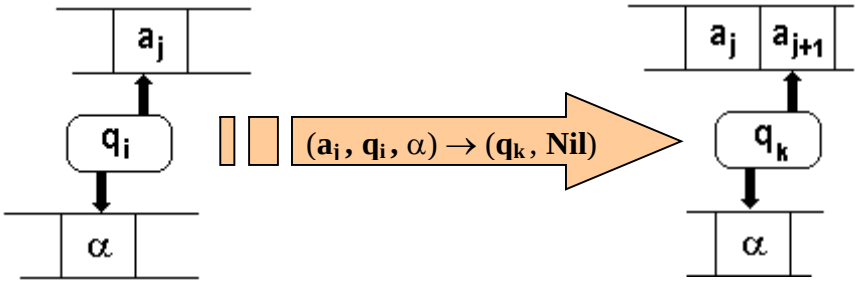
- La tête de lecture se déplace par examen des règles de transition de l'automate en y rajoutant l'examen du sommet de la pile. Le triplet ($\mathbf{a_j}$, $\mathbf{q_i}$,α) enclenche le processus de recherche d'une transition possible dans la partie gauche de la liste des règles de transitions de μ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

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.	

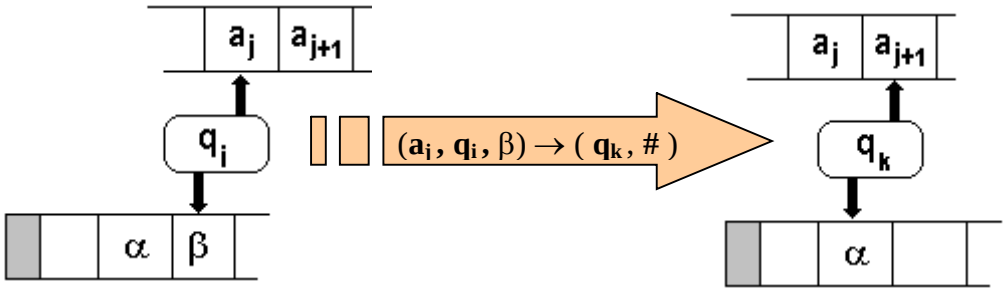
- **La transition $(a_j, q_i, \alpha) \rightarrow (q_k, x_n)$** signifie que l'automate peut passer de l'état $\mathbf{q_i}$ à l'état $\mathbf{q_k}$ à condition que le mot d'entrée débute par la chaîne préfixe $\mathbf{a_j}$ élément de $\mathbf{V_T}^*$ (notons que la chaîne $\mathbf{a_j}$ peut être réduite 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 :



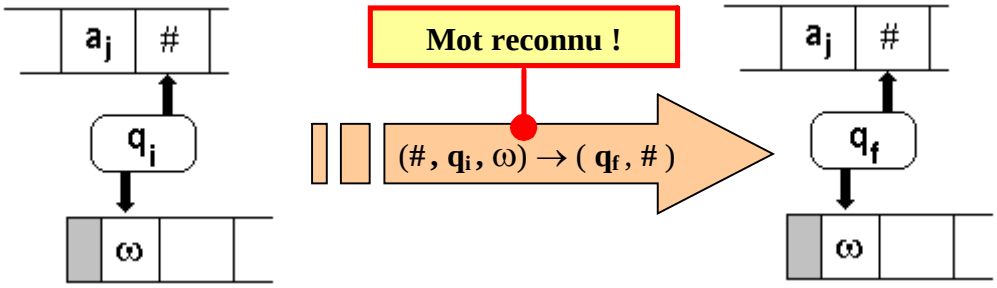
- **La transition $(a_j, q_i, \alpha) \rightarrow (q_k, \mathbf{Nil})$** signifie pour l'automate de ne rien faire dans la pile. Le résultat de la transition fait que l'état de l'automate passe de $\mathbf{q_i}$ à $\mathbf{q_k}$ et que la tête d'entrée pointe sur le symbole suivant de la bande d'entrée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



- **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.



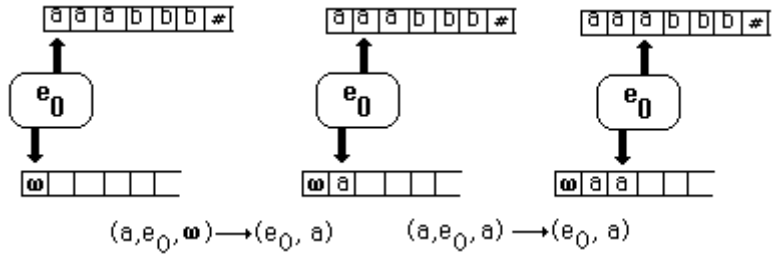
- Si la recherche de la transition μ : **(a_j , q_i ,** α) → .....ne donne pas de résultat on dit que l'auomate bloque : le mot n'est pas reconnu.

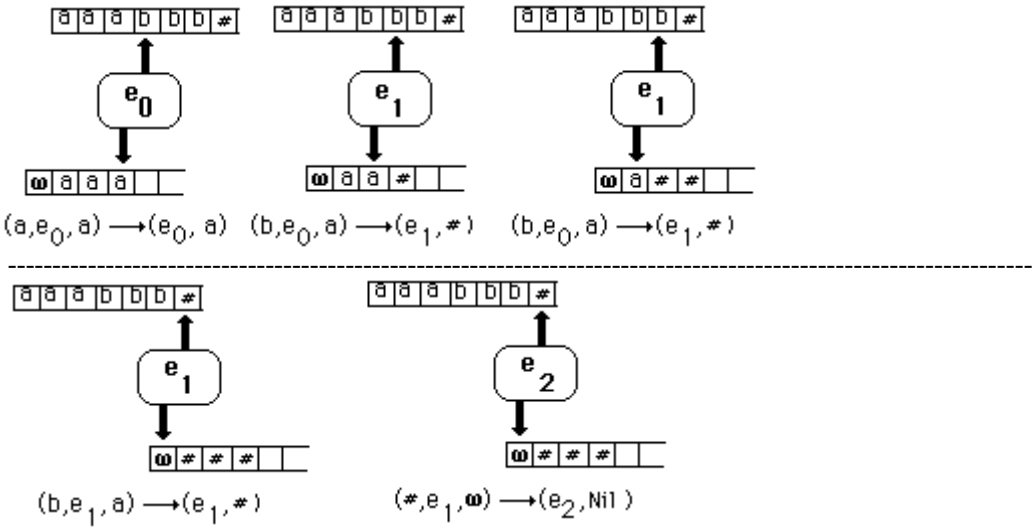
Exemple :

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 :

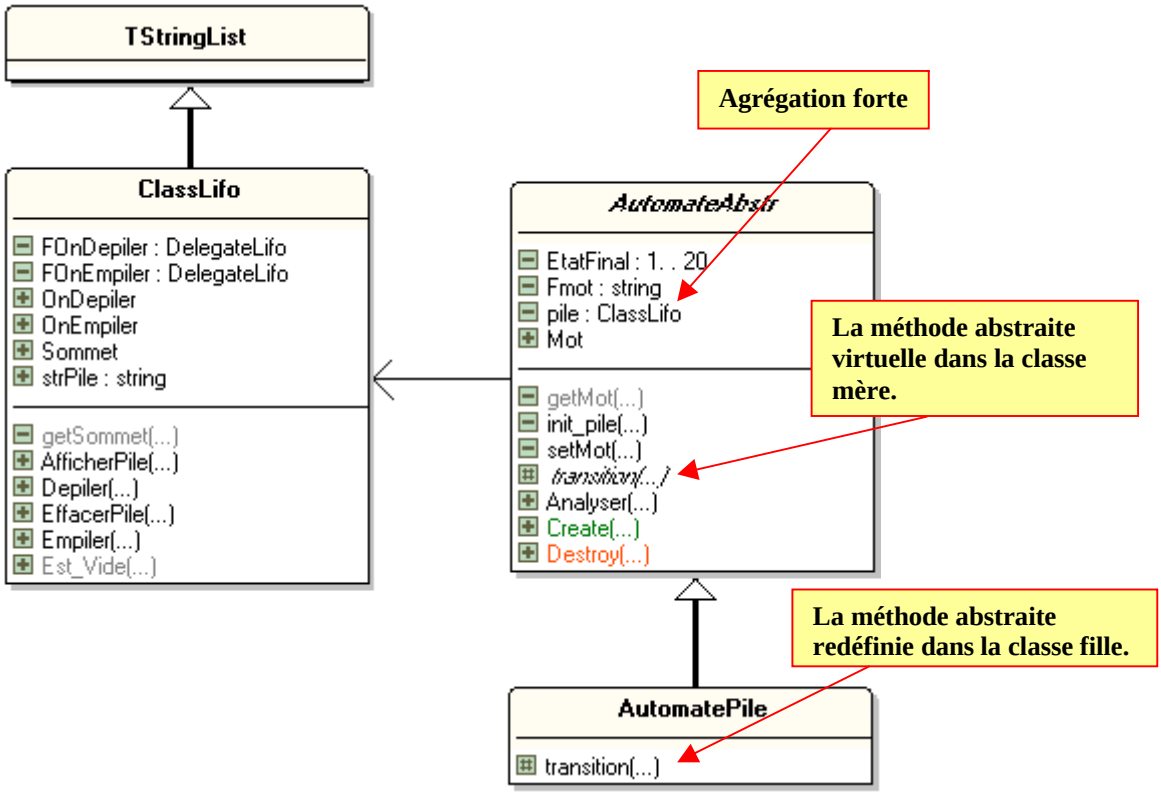
V_T = { **a** , **b** }
V_N = { S , A }
Axiome: S
Règles :
1 : S → **aSb**
2 : S → **ab**

2.3 Programme Delphi 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,** au lieu de la faire dériver du type **List** de Delphi, nous proposons de la dériver du type **TStringList** qui est une classe de liste de chaînes :

```
ClassLifo = class (TStringList)
 private
   FOnEmpiler : DelegateLifo ;
   FOnDepiler : DelegateLifo ;
   function getSommet:string;
 public
  strPile:string;
  function  Est_Vide  : boolean ;
  procedure  Empiler (elt : string ) ;
  procedure  Depiler (var elt : string ) ;
  procedure EffacerPile;
  procedure AfficherPile;
  property Sommet:string read getSommet;
  property  OnEmpiler : DelegateLifo  read  FOnEmpiler
           write  FOnEmpiler ;
  property  OnDepiler : DelegateLifo  read  FOnDepiler
           write  FOnDepiler ;
end;
```

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**.



Code complet de la classe pile de l'automate

```
unit  ULifoEvent ;

 interface
 uses  classes,Dialogs, SysUtils ;

 type
  DelegateLifo = procedure ( Sender: TObject ; s :string )of object ;

  PileVideException = class (Exception)
end;

ClassLifo = class (TStringList)
 private
  FOnEmpiler : DelegateLifo ;
  FOnDepiler : DelegateLifo ;
  function getSommet:string;
 public
  strPile:string;
  function  Est_Vide  : boolean ;
  procedure  Empiler (elt : string ) ;
  procedure  Depiler (var elt : string ) ;
  procedure EffacerPile;
  procedure AfficherPile;
  property Sommet:string read getSommet;
```

```
   property OnEmpiler : DelegateLifo read FOnEmpiler write FOnEmpiler;
   property OnDepiler : DelegateLifo read FonDepiler write FOnDepiler;
end;

implementation

 procedure ClassLifo.AfficherPile;
 var  i:integer;
 begin
  if self.count<>0 then
   for i:=0 to self.Count-1 do
    write(self.Strings[i],' ');
  writeln
 end;

 procedure  ClassLifo.Depiler(var elt : string ) ;
 begin
  if  not  Est_Vide  then
  begin
   elt :=self.Strings[0] ;
   strPile:='';
   self.Delete(0) ;
   if  assigned(FOnDepiler)  then
    FOnDepiler ( self ,elt )
  end
  else
   raise pilevideexception.create ('impossible de dépiler: pile vide' )
 end;

 procedure  ClassLifo.Empiler(elt : string ) ;
 begin
  self.Insert(0 , elt) ;
  if  assigned(FOnEmpiler)  then
   FOnEmpiler ( self ,elt )
 end;

 procedure ClassLifo.EffacerPile;
 begin
  self.Clear;
 end;

 function  ClassLifo.Est_Vide : boolean ;
 begin
  result :=  self.Count = 0 ;
 end;

 function ClassLifo.getSommet: string;
 begin
  result := self.Strings[0]
 end;

end.
```

Code complet des classes AutomateAbstr et AutomatePile

```
unit UAutomPile;

 interface

 uses ULifoEvent;
```

```
 type
  etat=-1..20;
  Vt=char;
  Vp=char;

  AutomateAbstr=class
 private
  EtatFinal:1..20;
  Fmot:string;
  pile:ClassLifo;
  procedure setMot(s:string);
  function getMot:string;
  procedure init_pile;
 protected
  procedure transition(ai:Vt;qj:etat;alpha:Vp; var qk:etat;
                var beta:vp); virtual;  abstract;
 public
  property Mot:string read getmot write setMot;
  procedure Analyser;
  constructor Create(fin:integer);
  destructor Destroy;override;
end;
AutomatePile=class(AutomateAbstr)
 protected
  procedure transition(ai:Vt;qj:etat;alpha:Vp; var qk:etat;
             var beta:vp);  override;
end;

implementation

 {---- AutomateAbstr ----}
const
 omega='$';
 non=-1;
 knil='@';

 constructor AutomateAbstr.Create(fin:integer);
 begin
  pile:=ClassLifo.Create;
  self.init_pile;
  if fin in [1..20] then
   EtatFinal:=fin
  else
   EtatFinal:=20;
  Fmot:='#';
 end;

 destructor AutomateAbstr.Destroy;
 begin
  pile.Free;
  inherited;
 end;

 function AutomateAbstr.getMot: string;
 begin
  result:= Fmot;
 end;

 procedure AutomateAbstr.setMot(s: string);
 var long:integer;
 begin
```

```
  long:=length(s);
  if long<>0 then
  begin
   if s[long]<>'#' then
    Fmot:=s+'#';
  end
  else
   fmot:='#';
end;

procedure AutomateAbstr.init_pile;
begin
 pile.EffacerPile;
 pile.Empiler(omega);
end;

procedure AutomateAbstr.Analyser;
var
 q:etat;
 numcar:integer;
 carPile:Vp;
 s:String;
 begin
  q:=0;
  numcar:=1;
  init_pile;
  while (q<>non)and(q<>EtatFinal) do
  begin
   transition(Fmot[numcar],q,pile.Sommet[1],q,carPile);
   pile.AfficherPile;
   numcar:=numcar+1;
   if carPile='#' then
    pile.Depiler(s)
   else
    if (carpile='a')or(carpile='b') then
     pile.Empiler(carPile);
  end;
  if (q=etatfinal)and(carpile=knil) then
   writeln('chaine reconnue !')
  else
   writeln('blocage, chaine non reconnue !')
 end;

 {---- AutomatePile ----}

procedure AutomatePile.transition(ai:Vt;qj:etat;alpha:Vp;
var qk:etat;
var beta:vp);
begin
 write('(',ai:2,',',qj:2,',',alpha:2,')');
 if (ai='a')and(qj=0)and(alpha=omega) then
 begin
  qk:=0;{ règle ; (a,e0,$) -->(e0,a) }
  beta:='a'
 end
 else
  if (ai='a')and(qj=0)and(alpha='a') then
  begin
   qk:=0; { règle ; (a,e0,a) -->(e0,a) }
   beta:='a'
  end
```

```
   else
    if (ai='b')and(qj=0)and(alpha='a') then
    begin
     qk:=1; { règle ; (b,e0,a) -->(e1,#) }
     beta:='#'
    end
    else
     if (ai='b')and(qj=1)and(alpha='a') then
     begin
      qk:=1; { règle ; (b,e1,a) -->(e1,#) }
      beta:='#'
     end
     else
      if (ai='#')and(qj=1)and(alpha=omega) then
      begin
       qk:=EtatFinal; { règle ; (#,e1,$) -->(e2,Nil) }
       beta:=KNil
      end
      else
      begin {blocage dans tous les autres cas}
       qk:=non;
       beta:=KNil
      end;
  write('--(',qk:2,',',beta,') pile=');
 end;

end.
```

Programme utilisant la classe Automate

```
program ProjAutomPile;

 {$APPTYPE CONSOLE}

uses
 SysUtils,
 UAutomPile in 'UAutomPile.pas';
var Automate:AutomatePile;

begin
 Automate:=AutomatePile.Create(2);
 Automate.Mot:='aaaaabbbbb';
 Automate.Analyser;
 readln;
end.
```

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

```
( a, 0, $)--( 0,a) pile=$
( a, 0, a)--( 0,a) pile=a $
( a, 0, a)--( 0,a) pile=a a $
( a, 0, a)--( 0,a) pile=a a a $
( a, 0, a)--( 0,a) pile=a a a a $
( b, 0, a)--( 1,#) pile=a a a a $
( b, 1, a)--( 1,#) pile=a a a $
( b, 1, a)--( 1,#) pile=a a $
( b, 1, a)--( 1,#) pile=a a $
( b, 1, a)--( 1,#) pile=a $
( #, 1, $)--( 2,@) pile=$
Chaine reconnue !
```

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.

6.2 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 pascal. 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,

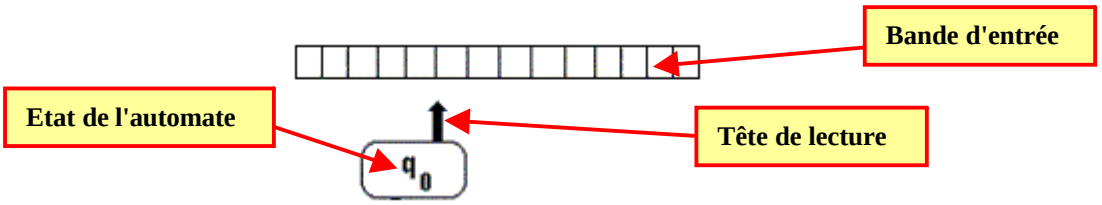
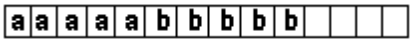


fig - un automate d'états finis

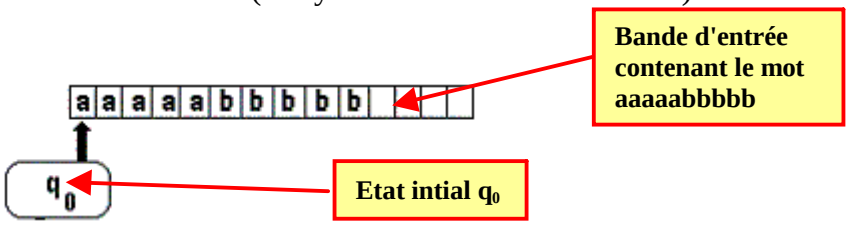
- Les règles de transitions spécifient la manipulation de la bande d'entrée de l'automate.

L'algorithme de fonctionnement d'un AEF :

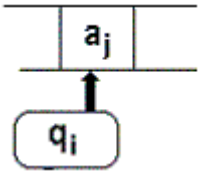
L'algorithme est très semblable à celui d'un automate à pile (en fait on peut considérer qu'il s'agit d'un cas particulier d'automate à pile dans lequel on n'effectue jamais d'action dans la pile), on fournit un mot que l'on écrit symbole par symbole de gauche à droite dans chaque case de l'automate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



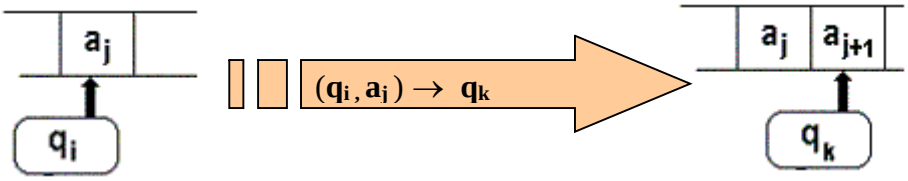
L'automate est mis à l'état initial $\mathbf{q_0}$, sa tête de lecture d'entrée est positionnée sur la première case à gauche de la bande d'entrée(1er symbole du mot à reconnaître) :



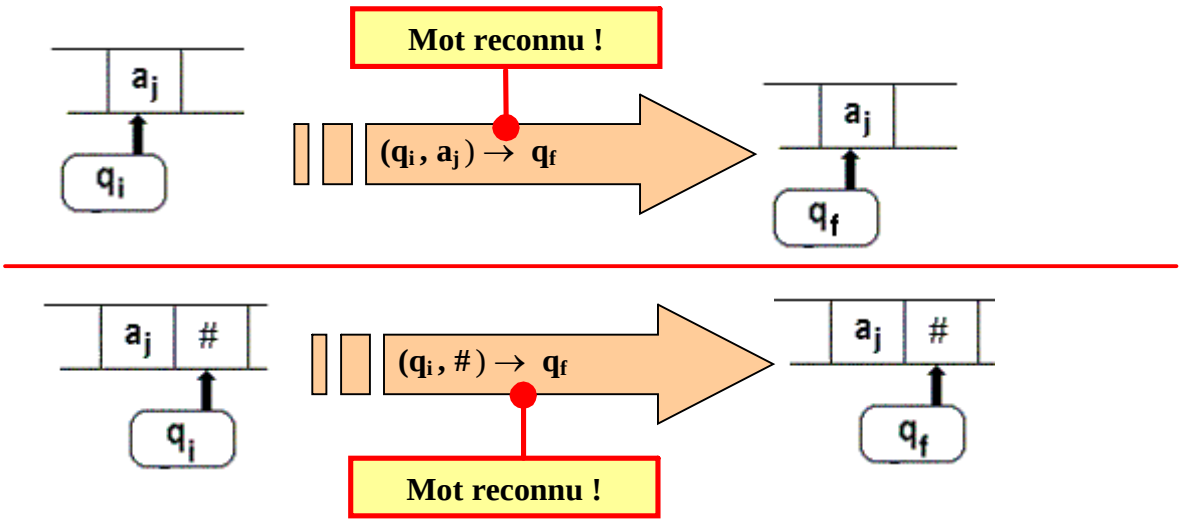
- La tête de lecture se déplace par examen des règles de transition de. Le couple $(\mathbf{a_j}, \mathbf{q_i})$ enclenche le processus de recherche d'une transition possible dans la partie gauche de la liste des règles de transitions de μ (il y a recherche de la transition μ : $(\mathbf{a_j}, \mathbf{q_i})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

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.



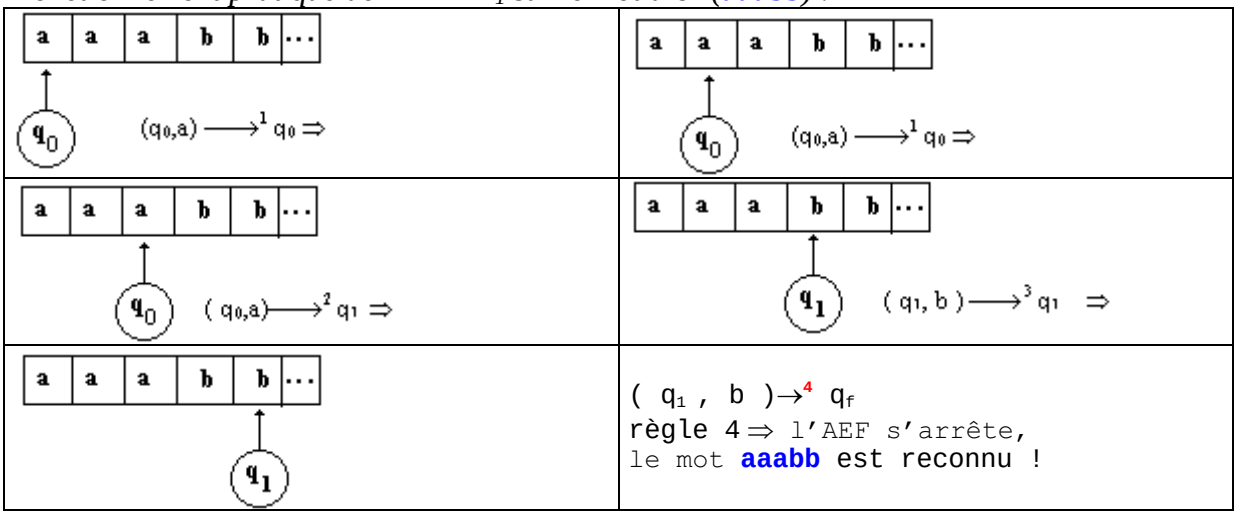
- Si l'AEF ne trouve pas de règle de transition commençant par (q_j , a_i), c'est à dire que le couple (q_j , a_i) n'a pas d'image par la fonction de transition μ, on dit alors que l'automate **bloque** : le mot n'est pas reconnu comme appartenant au langage.

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 : $S \rightarrow aS$ 2 : $S \rightarrow aA$ 3 : $A \rightarrow bA$ 4 : $A \rightarrow b$	$V_T = \{a,b\}$ $E = \{q_0 , q_1 , qf \}$ $\mu : (q_0, a) \rightarrow q_0$ $\mu : (q_0, a) \rightarrow q_1$ $\mu : (q_1, b) \rightarrow q_1$ $\mu : (q_1, b) \rightarrow q_f$ *(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 :

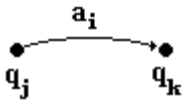
AEF A_1 non déterministe	AEF A_2 déterministe
$V_T = \{a,b\}$ $E = \{q_0 , q_1 , qf \}$ $\mu : (q_0, a) \rightarrow q_0$ $\mu : (q_0, a) \rightarrow q_1$ $\mu : (q_1, b) \rightarrow q_1$	$V_T = \{a,b,\#\}$ $E = \{q_0,q_1, q_2,qf \}$ $\mu : (q_0, a) \rightarrow q_1$ $\mu : (q_2, b) \rightarrow q_2$ $\mu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

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

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 !

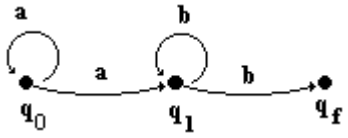
1.4 Graphe d'un automate déterministe ou non

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

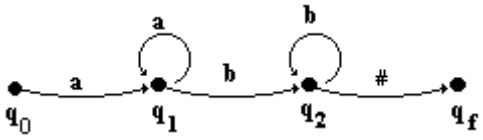
$(q_j , a_i) \rightarrow q_k$ est représentée par l'arc à 2 sommets : 

Exemples du graphe des deux AEF précédents :

graphe de A_1 :



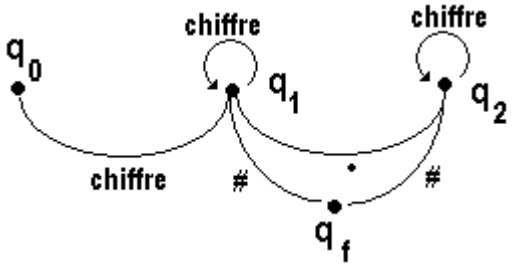
graphe de A_2 :



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 : $(q_0 , \mathbf{0})$,…, $(q_0 , \mathbf{9})$

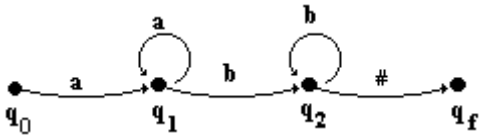
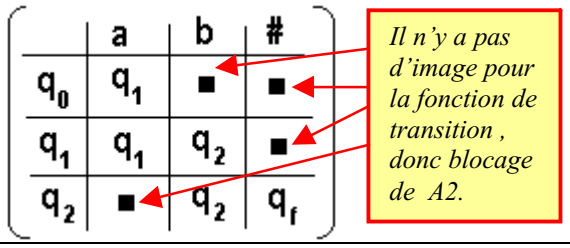
On peut voir à travers ce dernier exemple, le schéma général d'un analyseur lexical qui constitue la première étape d'un compilateur. Il suffit dès que le symbole a été reconnu donc à partir d'un état final $\mathbf{q_f}$ de repartir à l'état q_0.

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
	

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

Il n'y a pas d'image pour la fonction de transition , donc blocage de A2.

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 $\{a^n b^2 c^p / n \leq 0 \text{ et } p \leq 0\}$ 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.

1. Implantation d'une classe d'AEFD en Delphi

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.

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 AutomateEF, dont voici ci-dessous une écriture fictive pour une seule règle.

```
function AutomateEF.transition(q:T_etat;car:Vt):T_etat ;
begin
  if (q= qi)and(car= ak) then q:= qi {la règle (qj, ak) → qi}
  else ....
end ;
```

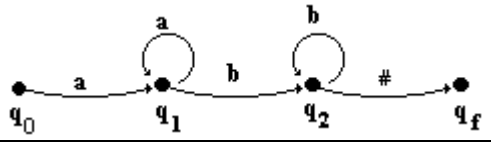
Toutes les autres règles sont implantées dans la **méthode** transition de la même façon.

L'appel de la **méthode** transition se fera à travers un objet AEFD de classe AutomateEF :

EtatApres := AEFD.transition(q_j, a_k) ; ***{la fonction renvoie l'état q_i}***

Exemple : morceaux de code

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




```
const
 imposs=-1;
 fin=20;
 finmot='#';
type
 T_etat=imposs..fin;
 Vt=char;
....
AutomateEF = class
  q : T_etat;
  mot:string;
  function transition(q:T_etat;car:Vt):T_etat;
end;
```

```
Implementation

function AutomateEF.transition(q:T_etat;car:Vt):T_etat;
{par les règles de transition :}
begin
  if (q=0)and(car='a') then q:=1 {(q0,a) → q1}
  else if (q=1)and(car='a') then q:=1 {(q1,a) → q1}
  else if (q=1)and(car='b') then q:=2 {(q1,b) → q2}
  else if (q=2)and(car='b') then q:=2 {(q2,b) → q2}
  else if (q=2)and(car=finmot) then q:=fin {(q2,#) → qf}
  else q:=imposs; {blocage, le caractère n'est pas dans Vt}
  result :=q
end;
```

Appel de la méthode transition dans le programme principal :

```
{Analyse}

numcar:=1;
etat:=0;
while (etat<>imposs)and(etat<>fin) do
begin
  Symsuiv(numcar,symlu); {fournit dans symlu le symbole suivant}
  etat:= AEFD.transition(etat,symlu);
end;
```

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** Delphi (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 AutomateEF).

Version réduite initiale du code : morceaux de code

```
const
 imposs=-1;
 fin=20;
 finmot='#';
type
 T_etat=imposs..fin;
 Vt=char;
 T_transition=array[T_etat,char] of T_etat;
```

```
AutomateEF = class

 q : T_etat;
 mot:string;
 table:T_transition; {matrice des transitions}

end;
```

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 du code : morceaux de code

```
AutomateEF = class
  q : T_etat;
  mot:string;
  table:T_transition; {matrice des transitions}
  procedure init_table;
  function transition(q:T_etat;car:Vt):T_etat;
end;

implementation

procedure AutomateEF.init_table;
{initialisation de la table des transitions}

var
 i:T_etat;
 j:0..255;
 k:char;
```

```
begin {init_table}
  for i:=imposs to fin do
    for j:=0 to 255 do
        Chargementde(table)
   .....
end; {init_table}

function AutomateEF.transition(q:T_etat;car:Vt):T_etat;
{par la table de transition :}
begin
  q := table[q,car];
  result := q;
end;
```

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

EtatApres := AEFD.transition(q_j, a_k) ; ***{la fonction renvoie l'état q_i}***

Exemple : le même 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

```
const
 imposs=-1;
 fin=20;
 finmot='#';
type
 T_etat=imposs..fin;
 Vt=char;
....
AutomateEF= class
 private
   EtatFinal : T_etat;
   Fmot:string;
   table:T_transition; {matrice des transitions}
  procedure init_table;
  function transition(q:T_etat;car:Vt):T_etat;
end;

Implementation

procedure AutomateEF.init_table;
{initialisation de la table des transitions}
var
 i:T_etat;
 j:0..255;
 k:char;
```

```
 begin {init_table}
    for i:=imposs to fin do
      for j:=0 to 255 do
        table[i,chr(j)]:=imposs;
   {par défaut tout est non reconnu}
    table[0,'a']:=1;
    table[1,'a']:=1;
    table[1,'b']:=2;
    table[2,'b']:=2;
    table[2,finmot]:=EtatFinal
end; {init_table}



 function AutomateEF.transition(q:T_etat;car:Vt):T_etat;
 {par la table de transition :}
 begin
   q := table[q,car];
   result := q;
 end;
```

Appel de la méthode transition dans le programme principal :

```
{Analyse}

numcar:=1;
etat:=0;
while (etat<>imposs)and(etat<>fin) do
begin
  Symsuiv(numcar,symlu); {fournit dans symlu le symbole suivant}
  etat:= AEFD.transition(etat,symlu);
end;
```

La unit Delphi contenant une classe abstraite d'automate et une classe fille d'AEF Reconnaissant le langage $a^n b^p$

```
Unit UautomEF;

interface
const
 imposs=-1;
 fin=20;
 finmot='#';
type
 T_etat=imposs..fin;
 Vt=char;
 T_transition=array[T_etat,char] of T_etat;

AutomateAbstr = class
 private
   Fmot:string;
   table:T_transition; {matrice des transitions}
   procedure setMot(s:string);
   function getMot:string;
   function transition(q:T_etat;car:Vt):T_etat;
   procedure Symsuiv (n: integer; var car:Vt);
protected
    EtatFinal : T_etat;
   procedure init_table; virtual;  abstract;
public
  property Mot:string read getmot write setMot;
  procedure Analyser;
  constructor Create(fin:integer);
end;

AutomateEF=class(AutomateAbstr)
 protected
    procedure init_table; override;
end;

Implementation
{--- AutomateAbstr ---}
constructor AutomateAbstr.Create(fin:integer);
 begin
    if fin in [1..20] then
     EtatFinal:=fin
   else
     EtatFinal:=20;
   Fmot:='#';
 init_table;
 end;

 function AutomateAbstr.getMot: string;
 begin
    result := Fmot;
 end;

procedure Symsuiv (n: integer; var car:Vt);
begin
 car := Fmot[n];
end;
```

```
procedure AutomateAbstr.setMot(s: string);
 var long : integer;
 begin
    long:=length(s);
    if long<>0 then   begin
      if s[long]<>'#' then
        Fmot:=s+'#';
    end
    else
     Fmot := '#';
 end;

function AutomateAbstr.transition(q:T_etat;car:Vt):T_etat;
    {par la table de transition :}
begin
      q := table[q,car];
      result := q;
end;

procedure AutomateAbstr.Analyser;
var
 etat : T_etat;  numcar : integer;
  s : string;  symlu : Vt;
begin
   numcar:=1;
   etat:=0;
   while (etat<>imposs)and(etat<> EtatFinal) do
   begin
      Symsuiv (numcar , symlu);
      numcar:= numcar+1;
    {fournit dans symlu le symbole suivant}
    etat:= self.transition(etat,symlu);
   end;
   if etat =etatfinal then
     writeln('chaine reconnue !')
   else
    writeln('blocage, chaine non reconnue !')
end;
{--- AutomateEF ---}
 procedure AutomateEF.init_table;
 {initialisation de la table des transitions}
 var  i:T_etat;  j:0..255;  k:char;
 begin
   for i:=imposs to fin do
         for j:=0 to 255 do
           table[i,chr(j)]:=imposs;
         {par défaut tout est non reconnu}
    table[0,'a']:=1;
    table[1,'a']:=1;
    table[1,'b']:=2;
    table[2,'b']:=2;
    table[2,finmot]:= EtatFinal;
 end;
end.
```

Programme utilisant la classe AutomateEF

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

```
program ProjAutomEF;

 {$APPTYPE CONSOLE}

uses
 SysUtils,
 UAutomEF in 'UAutomEF.pas';
var AEFD:AutomateEF;

 begin
   AEFD:= AutomateEF.Create(3);
   AEFD.Mot:='aaaaabbbbb';
   AEFD.Analyser;
   readln;
 end.
```

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

```
( 0, a)--->  1
( 1, a)--->  1
( 1, a)--->  1
( 1, a)--->  1
( 1, a)--->  1
( 1, b)--->  2
( 2, b)--->  2
( 2, b)--->  2
( 2, b)--->  2
( 2, b)--->  2
( 2, #)--->  3
chaine reconnue !
```

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

```
( 0, a)--->  1
( 1, a)--->  1
( 1, a)--->  1
( 1, a)--->  1
( 1, a)--->  1
( 1, b)--->  2
( 2, b)--->  2
( 2, a)---> -1
blocage, chaine non reconnue !
```

Nous remarquons dans ce dernier paragraphe, que nous avons mis en place un procédé général qui permet de construire méthodiquement en Delphi 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 Delphi 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** :

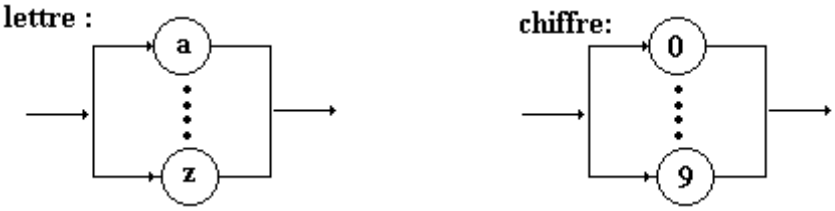
```
AutomateEF = class ( AutomateAbstr )
 protected
   procedure init_table; override;
end;

Implementation
```

```
(* --- AutomateEF
       Reconnaissant le langage L = { a^n b^p / (n ≤ 1) et (p ≤ 1) }
 ---*)
procedure AutomateEF.init_table;
{initialisation de la table des transitions}
var  i:T_etat;  j:0..255;  k:char;
begin
  for i:=imposs to fin do
       for j:=0 to 255 do
          table[i,chr(j)]:=imposs;
        {par défaut tout est non reconnu}
   table[0,'a']:=1;
   table[1,'a']:=1;
   table[1,'b']:=2;
   table[2,'b']:=2;
   table[2,finmot]:= EtatFinal;
end;
```

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 pascal-like

On donne des diagrammes syntaxiques de construction des identificateurs Pascal :
identificateur :



Construisons un programme Delphi 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 Delphi 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 déterministe.

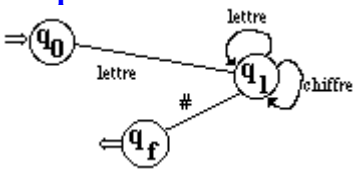
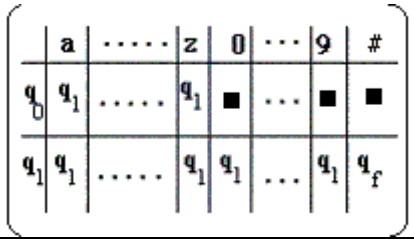
Construction de l'automate associé à G_{id}

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:	**Règles de l'automate associé à G_{id} :**
⟨*identificateur*⟩ ⇔ q_0 ⟨A⟩ ⇔ q_1 Etat initial = q_0 Etat final = q_f **Vocabulaire terminal de l'automate :** V_T' = V_T ∪ {#} = Chiffre ∪ Lettre ∪ {#}	(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:**
	

Programme associé à l'automate

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

```
AutomateEF = class ( AutomateAbstr )
 protected
   procedure init_table; override;
end;

implementation

procedure AutomateEF.init_table;
 {initialisation de la table des transitions}
 var  i:T_etat;  j:0..255;  x:char;
 begin
   for i:=imposs to fin do
     for j:=0 to 255 do
       table[i,chr(j)]:=imposs;
   for x:='a' to 'z' do
    begin
      table[0,x]:=1;  { (q0,lettre) → q1 }
      table[1,x]:=1;  { (q1,lettre) → q1 }
    end;
    for x:='0' to '9' do
      table[1,x]:=1;  { (q1,chiffre) → q1 }
    table[1,finmot]:= EtatFinal;  { (q1,#) → qf }
end;
```

Programme utilisant la classe AutomateEF
Reconnaissant le langage des identificateurs

```
program ProjAutomEF;

 {$APPTYPE CONSOLE}

uses
 SysUtils,
 UAutomEF in 'UAutomEF.pas';
var AEFD:AutomateEF;

 begin
   AEFD:= AutomateEF.Create(2);
   AEFD.Mot:= 'v14bcd73' ;
   AEFD.Analyser;
   readln;
 end.
```

*Exécution de ce programme sur l'identificateur **prix2** :*

```
( 0, p)-->  1
( 1, r)-->  1
( 1, i)-->  1
( 1, x)-->  1
( 1, 2)-->  1
( 1, #)-->  2
chaine reconnue !
```

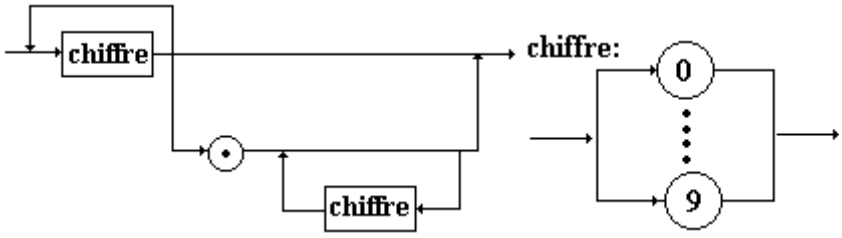
*Exécution de ce programme sur l'exemple **v14bcd73** :*

```
( 0, v)-->  1
( 1, 1)-->  1
( 1, 4)-->  1
( 1, b)-->  1
( 1, c)-->  1
( 1, d)-->  1
( 1, 7)-->  1
( 1, 3)-->  1
( 1, #)-->  2
chaine reconnue !
```

3.4 Exemple : les constantes numériques

On donne des diagrammes syntaxiques de construction des constantes décimales positives Pascal-like :

constante :



Construisons un programme Delphi 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 pascal 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

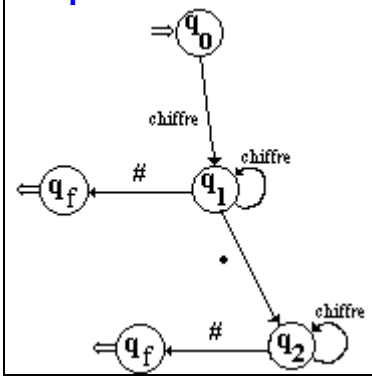
Etats associés aux éléments de V_N:
⟨*constante*⟩ ⇔ q_0
⟨ A ⟩ ⇔ q_1
⟨ B ⟩ ⇔ q_2
Etat initial = q_0
Etat final = q_f

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

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

```
AutomateEF = class ( AutomateAbstr )
 protected
   procedure init_table; override;
end;

implementation

procedure AutomateEF.init_table;
 {initialisation de la table des transitions}
 var  i:T_etat;  j:0..255;  x:char;
 begin
   for i:=imposs to fin do
     for j:=0 to 255 do
       table[i,chr(j)]:=imposs;
   for x:='0' to '9' do
    begin
      table[0,x]:=1;  { (q0,chiffre) → q1  }
      table[1,x]:=1;  { (q1,chiffre) → q1  }
      table[2,x]:=2;  { (q2,chiffre) → q2  }
    end;
    table[1,'.']:=2;   { (q1,'.') → q2  }
    table[1,finmot]:= EtatFinal;   { (q1,#) →  qf  }
    table[2,finmot]:= EtatFinal;   { (q2,#) →  qf  }
end;
```

Programme utilisant la classe AutomateEF

Reconnaissant le langage des constantes numériques

```
program ProjAutomEF;

 {$APPTYPE CONSOLE}

 uses
  SysUtils,
  UAutomEF in 'UAutomEF.pas';
 var AEFD:AutomateEF;

 begin
   AEFD:= AutomateEF.Create(3);
   AEFD.Mot:= 145.78 ;
   AEFD.Analyser;
   readln;
 end.
```

*Exécution de ce programme sur l'exemple **145** :*

```
( 0, 1)--->  1
( 1, 4)--->  1
( 1, 5)--->  1
( 1, #)--->  3
chaine reconnue !
```

*Exécution de ce programme sur l'exemple **145.78** :*

```
( 0, 1)--->  1
( 1, 4)--->  1
( 1, 5)--->  1
( 1, .)--->  2
( 2, 7)--->  2
( 2, 8)--->  2
( 2, #)--->  3
chaine 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.

6.3 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

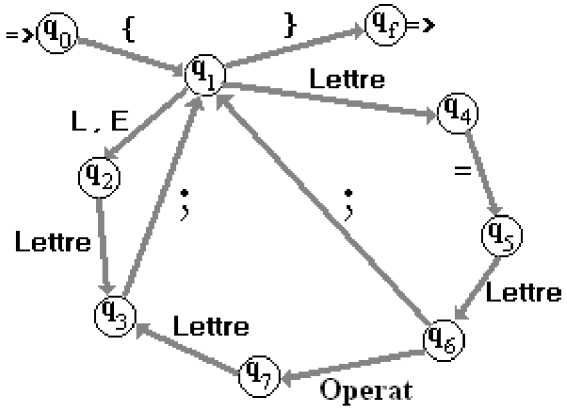
Soit l'AEFD A, $A = (V_T', E, q_0, q_f, \mu)$
$E = \{ \mathbf{q_0}, q_1, q_2, q_3, q_4, q_5, q_6, q_7, \mathbf{q_f} \}$
état initial : $\mathbf{q_0}$
état final : $\mathbf{q_f}$
V_T = { **'a'** , **'b'**,..., **'z'** , **'}'** , **'{'** , **';'** , **'L'**, **'E'** , **'+'** , **'='**, **'*'** , **'-'**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

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 :

```
const
 imposs=-1;  fin=20;  finmot='#';
type
   T_etat=imposs..fin;
   Vt=char;
   T_transition=array[T_etat,char] of T_etat;

AutomateEF = class ( AutomateAbstr )
 protected
    procedure init_table; override;
end;

implementation

procedure AutomateEF.init_table;
 {initialisation de la table des transitions}
 var  i:T_etat;  j:0..255;  k:char;
 begin
   for i:=imposs to fin do
     for j:=0 to 255 do
        table[i,chr(j)]:=imposs;
```

```
   table[0,'{']:=1;
   table[1,'}']:=fin;
   for k:='a' to 'z' do
   begin
     table[1,k]:=4;
     table[2,k]:=3;
     table[5,k]:=6;
     table[7,k]:=3;
   end;
   table[1,'E']:=2;
   table[1,'L']:=2;
   table[3,';']:=1;
   table[4,'=']:=5;
   table[6,'+']:=7;
   table[6,'*']:=7;
   table[6,'-']:=7;
   table[6,'%']:=7;
   table[6,'/']:=7;
   table[6,';']:=1;
end;
```

Programme d'analyseur utilisant la classe AutomateEF

Reconnaissant le langage L(**G**)

```
program ProjAutomEF;

 {$APPTYPE CONSOLE}

uses
 SysUtils,
 UAutomEF in 'UAutomEF.pas';
var AEFD:AutomateEF;

 begin
   AEFD:= AutomateEF.Create(8);
   AEFD.Mot:= '{La;Lb;Ea;a=b*c;}';
   AEFD.Analyser;
   readln;
 end.
```

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

```
( 0, {)--> 1
( 1, L)--> 2
( 2, a)--> 3
( 3, ;)--> 1
( 1, L)--> 2
( 2, b)--> 3
( 3, ;)--> 1
( 1, E)--> 2
( 2, a)--> 3
( 3, ;)--> 1
( 1, a)--> 4
( 4, =)--> 5
( 5, b)--> 6
( 6, *)--> 7
( 7, c)--> 3
( 3, ;)--> 1
( 1, })--> 8
chaine reconnue !
```

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

```
( 0, {)--> 1
( 1, L)--> 2
( 2, a)--> 3
( 3, ;)--> 1
( 1, L)--> 2
( 2, b)--> 3
( 3, ;)--> 1
( 1, E)--> 2
( 2, a)--> 3
( 3, =)--> -1
blocage, chaine non reconnue !
```

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

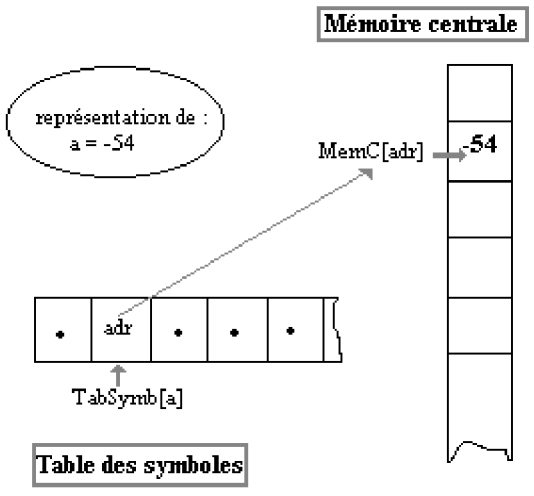
Rappelons les spécifications proposées par l'énoncé :

Lx correspond à **lire**(x)
Ex correspond à **ecrire**(x)

x = y correspond à **x** ← y
a,**b**,..., **z** correspondent à des variables 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.

Le mécanisme d'accès est simple



Spécifications opérationnelles

L'énoncé nous propose une spécification d'implantation en Delphi 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.

Implantation Delphi de ces spécifications de données

```
const
 adresse=0..maxadresse;
type
 Symbole=char;
 T_mem=array[adresse] of integer;
 T_symb=array[Symbole]of adresse;
var
 MemC:T_mem; // la mémoire centrale
 TabSymb:T_symb; // la table des symboles
```

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 Delphi.

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 ; readln(MemC[TabSymb[x]])

Interprétation de l'instruction E :

Ex	≅	adressecourante := TabSymb[x]; writeln(MemC[adressecourante])

Interprétation de l'instruction x = y :

x = y	≅	MemC[TabSymb[x]]:= MemC[TabSymb[y]]

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

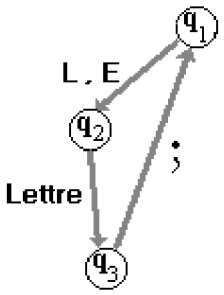
`x=y oper z` ≅ ***oper** est l'un des opérateurs suivants :* *{ '+', '*', '-', '/' }*	MemC[TabSymb[x]]:= MemC[TabSymb[y]] **oper** MemC[TabSymb[z]]

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

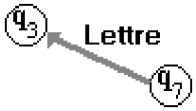
Ainsi, nous passerons de la version analyseur à la version interpréteur en suivant les chemins du graphe et en repérant les points clefs où l'interprétation d'une instruction est possible. Nous adoptons comme stratégie le fait que le lancement d'une interprétation ne sera possible que lorsque l'on sera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



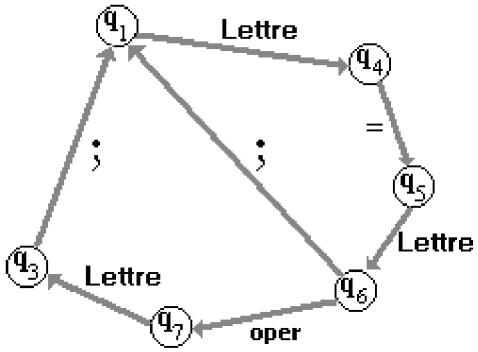
Nous voyons qu'il n'y a que deux manières différentes d'arriver en q_3 : soit en venant de q_2, soit en venant de q_7.

> 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-Delphi le code de lancement de l'interprétation de Lx ou de Ex.
 	
Ce code est à rajouter à l'AEFD précédent.
	If (etat=q_3)and(etatavant=q_2)**and**(symprec=L)**then**
	begin *{on interprète le Lx}*
	TabSymb[symlu] := adressecourante ;
	adressecourante := adressecourante +1 ;
	readln(MemC[TabSymb[symlu]])
	end
	else *{on interprète le Ex}*
	begin
	adressecourante := TabSymb[symlu];
	writeln(MemC[adressecourante])
	end

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 : $q_1 \rightarrow q_4 \rightarrow q_5 \rightarrow q_6 \rightarrow q_7 \rightarrow q_3$. 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**) → 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 $q_1 \rightarrow q_4$)
y est connu lorsque l'AEFD est à l'état q_6 (à la fin de l'arc $q_5 \rightarrow q_6$)
z est connu lorsque l'AEFD est à l'état q_3 (à la fin de l'arc $q_7 \rightarrow q_3$)

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 :

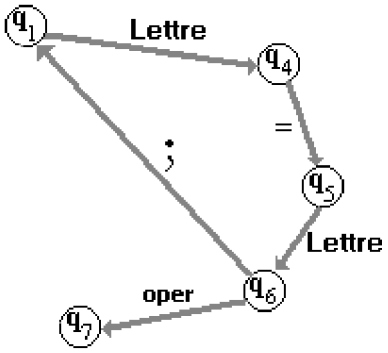
- z est stocké dans **symlu**, il est reconnu à l'état q_3.
- y est stocké dans **symprec**, il est reconnu à l'état q_6.
- x est stocké dans **symavant**, il est reconnu à l'état q_4.
- l'opérateur **oper** est reconnu à l'état q_7,il est alors rangé dans une variable **OperVar** de type Vt servant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

Voici donc en pseudo-Delphi 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 then symavant := symlu ;
If etat=q6 then symprec := symlu ;
If etat=q7 then
  If symlu in Operat then   OperVar := symlu;
If (etat=q3)and(etatavant=q7)then {on interprète x = y oper z}
case OperVar of
 '+':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]+MemC[TabSymb[symlu]];
 '-':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]-MemC[TabSymb[symlu]];
 '*':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]*MemC[TabSymb[symlu]];
 '/':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]div MemC[TabSymb[symlu]];
 '%':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]mod MemC[TabSymb[symlu]];
end;
```

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.

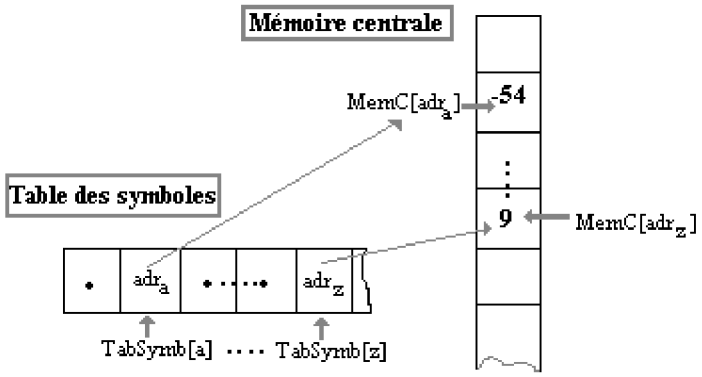
Le chemin d'analyse d'une instruction x=y est :
$q_1 \rightarrow q_4 \rightarrow q_5 \rightarrow q_6 \rightarrow q_1$.
Comme dans l'analyse du problème précédent nous avons :

- **;** 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-Delphi, 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)and(etatavant=q6)then {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 Delphi d'une classe d'interpréteur **InterpreteurLang** permettant d'étendre l'analyseur en interpréteur :

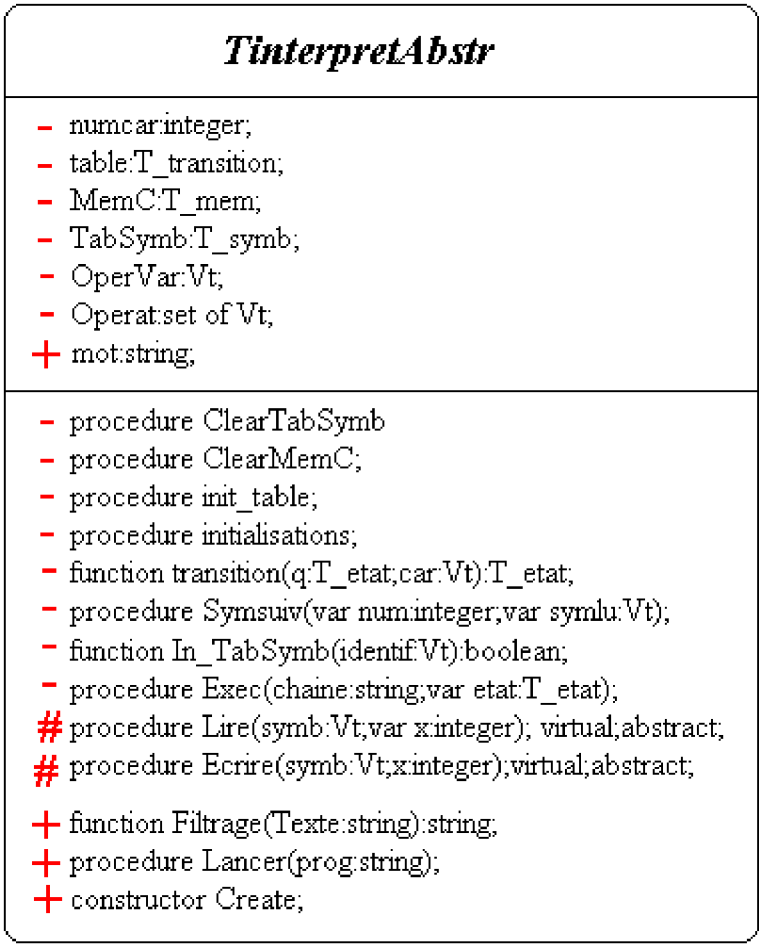
```
type
  T_etat=imposs..fin;    Vt=char;

procedure InterpreteurLang.init_table;
{initialisation de la table des transitions de l'automate AEFD }

procedure InterpreteurLang.ClearMemC;
 // RAZ mémoire centrale

procedure InterpreteurLang.ClearTabSymb;
 // RAZ table des symboles

 procedure InterpreteurLang.initialisations;
 // RAZ tout

{---------------------- INTERPRETEUR -------------------}

procedure InterpreteurLang.Interprete(chaine:string;var etat:T_etat);
{moteur d'analyse et d'interpretation de l'automate}

 function InterpreteurLang.transition(q:T_etat;car:Vt):T_etat;
 {par la table de transition :}

procedure InterpreteurLang.Symsuiv(var num:integer;var symlu:Vt);
{fournit le symbole suivant à analyser}

function InterpreteurLang.In_TabSymb(identif:Vt):boolean;
{indique si le symbole identif est deja dans TabSymb}
```

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 construisons une classe abstraite **TinterpretAbstr** 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 **TinterpretAbstr**.

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

Nous reprenons toutes les procédures et fonctions du programme console et nous les transformons en méthodes de classe. Ci-dessous un diagramme UML-like de la signature de la classe **TinterpretAbstr**. Nous ajoutons à cette classe, en visibilité **protected,** les deux méthodes abstraites :

```
procedure Lire(symb:Vt;var x:integer); // symb = la variable à lire, x = sa valeur entière
procedure Ecrire(symb:Vt; x:integer);  // symb = la variable à ecrire, x = sa valeur entière
```



Nous reprenons dans une unit Delphi que nous nommons **Uinterpreteur,** les mêmes déclarations de constantes et de type que dans le programme console :

```
Unit Uinterpreteur :
    interface
    const
      imposs=-1;
      fin=8;
      finmot='#';
      maxadresse=100;
    type
     T_etat=imposs..fin;
     Vt=char;
     T_transition=array[T_etat,char] of T_etat;
     adresse=0..maxadresse;
     Symbole=char;
     T_mem=array[adresse] of integer;
     T_symb=array[Symbole]of adresse;
```

Ce qui nous donne dans **Uinterpreteur** la signature suivante pour la classe abstraite TinterpretAbstr :

```
  TinterpretAbstr=class
     private
      numcar:integer;
      table:T_transition;
      MemC:T_mem;
      TabSymb:T_symb;
```

```
    OperVar:Vt;
    Operat : set of Vt;
    EtatFinal : T_etat;
    procedure ClearTabSymb;
    procedure ClearMemC;
    procedure init_table;
    procedure initialisations;
    function transition(q:T_etat;car:Vt):T_etat;
    procedure Symsuiv(var num:integer;var symlu:Vt);
    function In_TabSymb(identif:Vt):boolean;
    procedure Exec(chaine:string;var etat:T_etat);
  protected
    procedure Lire(symb:Vt;var x:integer); virtual;abstract;
    procedure Ecrire(symb:Vt;x:integer); virtual;abstract;
  public
    mot:string;
    function Filtrage(Texte:string):string;
    procedure Lancer(prog:string);
    constructor Create(fin : T_etat) ;
  end;
```

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 :

function Filtrage(Texte: **string**): **string**;

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;}#
```

Nous donnons ici la partie **implementation** de la unit **Uinterpreteur** avec les corps des méthodes :

```
{ TinterpretAbstr }

procedure TinterpretAbstr.ClearMemC;
// RAZ mémoire centrale
var i:adresse;
begin
 for i:=0 to maxadresse do
  MemC[i]:=0;
end;
```

```
procedure TinterpretAbstr.ClearTabSymb;
// RAZ table des symboles
var i:Symbole;
begin
 for i:=chr(0) to chr(255) do
  TabSymb[i]:=0;  {0 indique : variable non definie}
end;

procedure TinterpretAbstr.init_table;
{initialisation de la table des transitions de l'automate AEFD }
var
  i:T_etat;   j:0..255;   k:char;
begin
 for i:=imposs to fin do
   for j:=0 to 255 do
   table[i,chr(j)]:=imposs;{par défaut tout est non reconnu}
 table[0,'{']:=1;
 table[1,'}']:= EtatFinal;
 for k:='a' to 'z' do
 begin
  table[1,k]:=4;
  table[2,k]:=3;
      table[5,k]:=6;
  table[7,k]:=3;
 end;
 table[1,'E']:=2;
 table[1,'L']:=2;
 table[3,';']:=1;
 table[4,'=']:=5;
 table[6,'+']:=7;
  table[6,'*']:=7;
 table[6,'-']:=7;
 table[6,'%']:=7;
 table[6,'/']:=7;
 table[6,';']:=1;
end;

procedure TinterpretAbstr.initialisations;
// RAZ tout
begin
  ClearMemC;
  ClearTabSymb;
  init_table;
  Operat :=['+', '-', '*', '/', '%' ]
end;

function TinterpretAbstr.transition(q: T_etat; car: Vt): T_etat;
{par la table de transition :}
begin
 result:=table[q,car];
end;

procedure TinterpretAbstr.Symsuiv(var num:integer;var symlu:Vt);
 {fournit le symbole suivant à analyser}
begin
  if num<=length(mot) then
  begin
   symlu:=mot[num];
   num:=num+1
```

(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

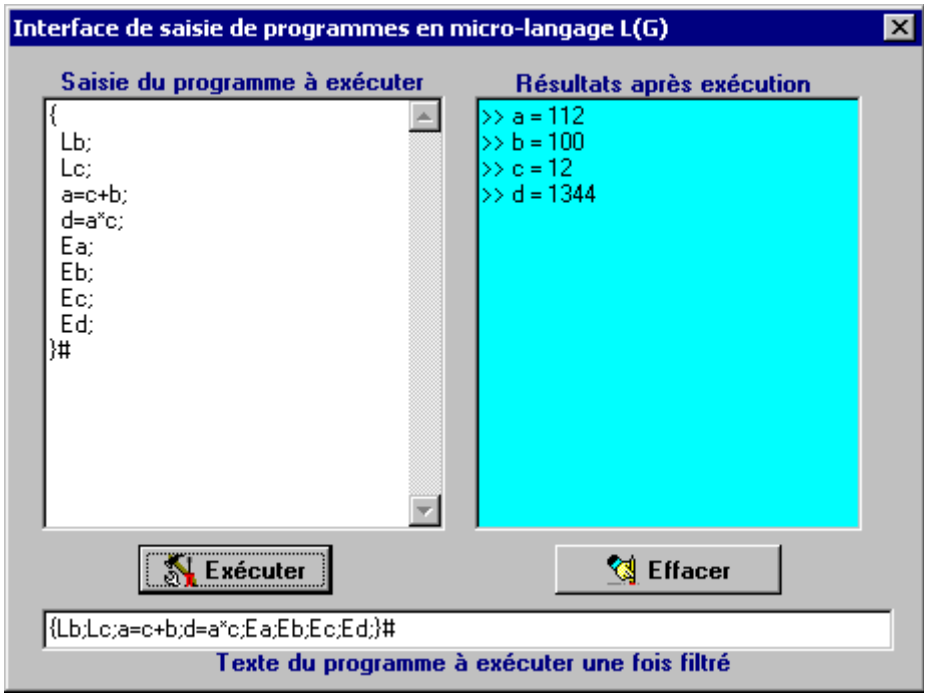
```
   end
   else         {si on veut lire apres la fin}
    symlu:=chr(0);{caractere NUL invisible     }
  end;

  function TinterpretAbstr.In_TabSymb(identif:Vt):boolean;
   {indique si le symbole identif est deja dans TabSymb}
  begin
    if TabSymb[identif]=0 then In_TabSymb:=false
    else In_TabSymb:=true
  end;

  procedure TinterpretAbstr.Exec(chaine: string; var etat: T_etat);
 {moteur d'analyse et d'interpretation de l'automate}
  var
     symlu:Vt;
     adressecourante:adresse;
     symprec,symavant:Vt;
     etavant:T_etat;
 begin {Interpreteur - Exec}
   numcar:=1;
   etat:=0;
   adressecourante:=1;  {on commence toujours a 1}
   mot:= chaine;
   while (etat<>impossible)and(etat<>fin) do
   begin
    Symsuiv(numcar,symlu);
    etavant:=etat;
    etat:=transition(etat,symlu);
    {------------ partie due a l'interpretation ------------}
    if (etat=2)then
      symprec:=symlu;
    if etat=4 then
    begin
      symavant:=symlu;
      if not In_TabSymb(symlu)then
      begin
       TabSymb[symlu]:=adressecourante;
       adressecourante:=adressecourante+1
      end
    end;
    if etat=6 then
      symprec:=symlu;
    if etat=7 then
     if symlu in Operat then
       OperVar := symlu;
    if (etat=3)and(etavant=2)and(symprec='L') then { Lx; }
    begin
      TabSymb[symlu]:=adressecourante;
      adressecourante:=adressecourante+1;
      Lire(symlu,MemC[TabSymb[symlu]]);
    end
    else
    if (etat=3)and(etavant=2)and(symprec='E') then { Ex; }
    begin
     adressecourante:=TabSymb[symlu];
     Ecrire(symlu,MemC[adressecourante]);
    end
    else
    if (etat=1)and(etavant=6)then              { x=y; }
```

```
    begin
     MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]
    end
    else
    if (etat=3)and(etavant=7)then              { x = y oper z; }
     case OperVar of
      '+':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]+MemC[TabSymb[symlu]];
      '-':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]-MemC[TabSymb[symlu]];
      '*':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]*MemC[TabSymb[symlu]];
      '/':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]div MemC[TabSymb[symlu]];
      '%':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]mod MemC[TabSymb[symlu]];
     end;
    {-------------------------------------------------------}
   end;
  end;{Interpreteur - Exec}

  function TinterpretAbstr.Filtrage(Texte: string): string;
  { filtrage du texte à interpréter :conservation des éléments
   du vocabulaire Vt et élimination des autres }
  var s:string;
     i:integer;
  begin
   s:='';
   for i:=1 to length(Texte) do
    if Texte[i] in ['a'..'z',';','{','}','L','E','=',finmot]+Operat  then
     s:=s+ Texte[i];
   result:=s
  end;


  procedure TinterpretAbstr.Lancer(prog: string);
  { uniquement pour appeler la méthode privée Exec
    et envoyer des messages à l'utilisateur selon
    la manière dont s'est passée l'exécution.
  }
  var q:T_etat;
  begin
   Exec(prog,q);
   if q=imposs then
     MessageDlg('Blocage, erreur de syntaxe !'+
     #13+#10+copy(prog,1,numcar)+'<<--', mtError ,[mbOk], 0)
   else
    if q=fin then
     MessageDlg('Exécution terminée !', mtInformation  ,[mbOk], 0);
  end;

 {----  le constructeur TinterpretAbstr ----}
 constructor TinterpretAbstr.Create(fin : T_etat) ;
 begin
    if fin in [1..20] then   EtatFinal:=fin
    else     EtatFinal:=20;
    initialisations;
 end;

end. {----  fin de la unit  ----}
```

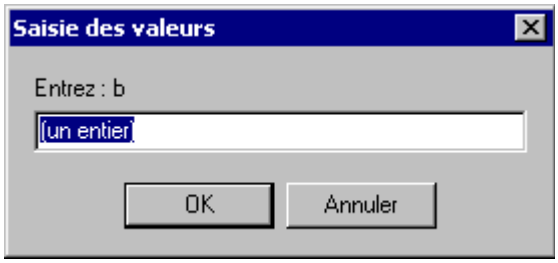
2°) Construction de deux classes héritant de TinterpretAbstr

Application IHM - première classe dérivée
Nous voulons écrire **une application IHM** utilisant la classe d'interpréteur que nous venons de construire, selon par exemple le modèle ci-dessous :

Nous afficherons les résultats par une **surcharge dynamiqu**e (redéfinition) de la méthode Ecrire à travers un TMemo :



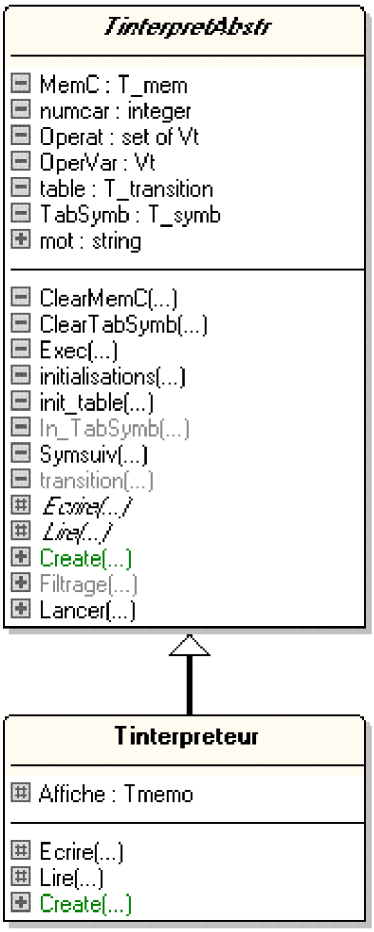
Nous saisirons les valeurs par une redéfinition de la méthode Lire à travers une inputBox :



Soit la classe Tinterpreteur héritant de notre classe abstraite, qui reçoit lors de la construction d'un objet d'interpréteur la référence d'un Tmemo déjà instancié dans l'IHM afin d''afficher les résultats :

```
{ Tinterpreteur cas de Delphi en mode application-IHM }
```

::UInterprete

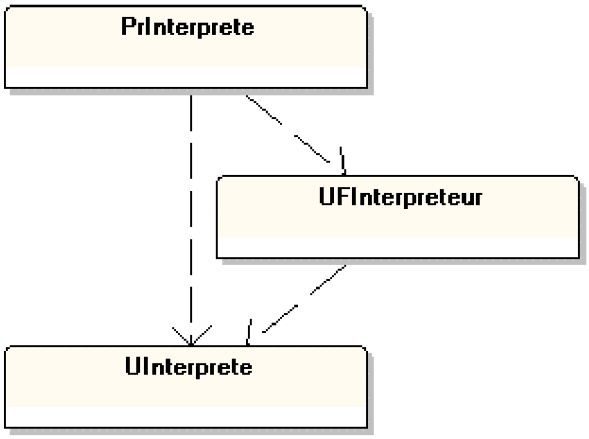


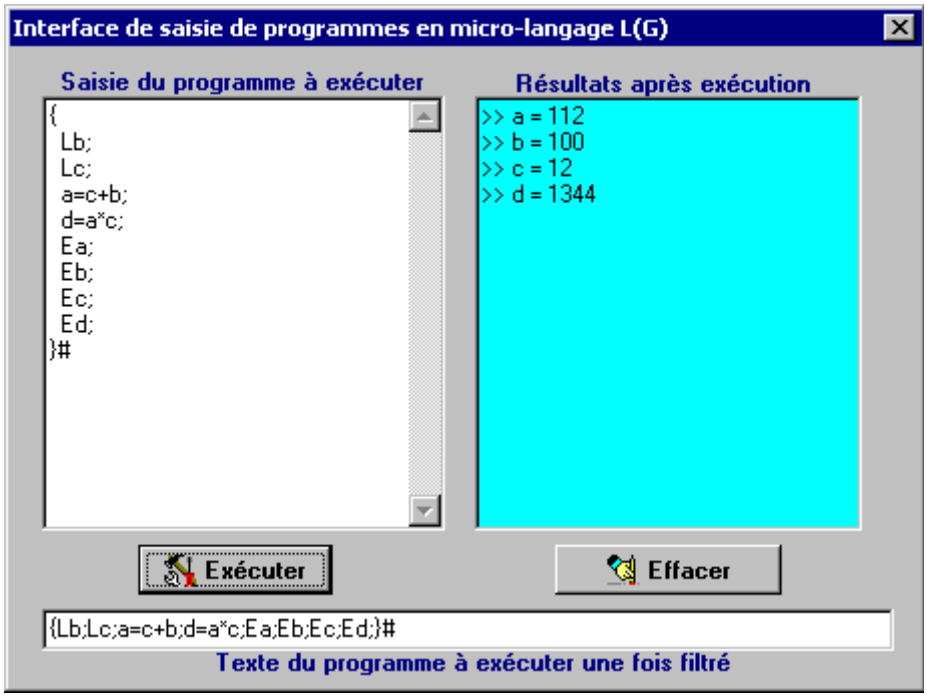

```
Tinterpreteur=class(TinterpretAbstr)
   protected
    Affiche:Tmemo;
    procedure Lire(symb:Vt;var x:integer); override;
    procedure Ecrire(symb:Vt;x:integer); override;
   public
    constructor Create(sortie:TMemo);
 end;
```

```
procedure Tinterpreteur.Ecrire(symb:Vt;x:integer);
begin
 Affiche.Lines.Append('>> '+symb+' = '+inttostr(x))
end;

procedure Tinterpreteur.Lire(symb:Vt;var x: integer);
var InputString:string;
begin
 InputString:= InputBox('Saisie des valeurs', 'Entrez : '+symb, '(un entier)');
 try
  x:=strtoint(InputString);
 except
  x:=1;
 end
```

```
end;

{----  le constructeur Tinterpreteur ----}
constructor Tinterpreteur.Create(sortie: TMemo);
begin
 inherited Create;
 Affiche:=sortie
end;
```

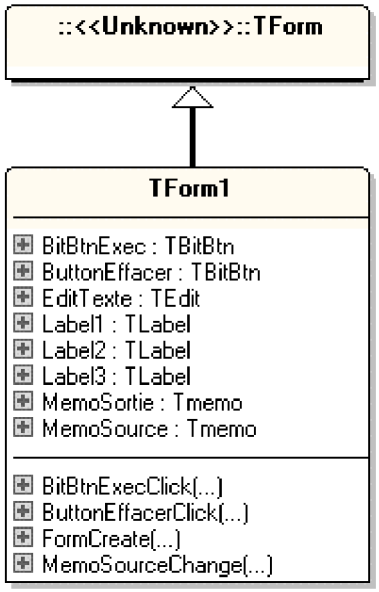
Terminons en livrant le code source et l'organisation de l'IHM de saisie et de traitement (exe et projet complet) :






```
unit UFInterpreteur;

interface

uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls,UInterprete, Buttons;

 ::UFInterpreteur
```




```
type
  TForm1 = class(TForm)
    MemoSource: TMemo;
    MemoSortie: TMemo;
    EditTexte: TEdit;
    BitBtnExec: TBitBtn;
    ButtonEffacer: TBitBtn;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure MemoSourceChange(Sender: TObject);
    procedure BitBtnExecClick(Sender: TObject);
    procedure ButtonEffacerClick(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;
  interpreteur:Tinterpreteur; { objet interpréteur }

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
{ instanciation de l'objet interpréteur
  et liaison avec le Tmemo MemoSortie
}
begin
 interpreteur:=Tinterpreteur.Create(MemoSortie);
```

```
 EditTexte.Text:= interpreteur.Filtrage(MemoSource.text);
end;

procedure TForm1.MemoSourceChange(Sender: TObject);
{ filtrage du texte sur l'événement OnChange du TMemo
  et stockage dans le TEdit EditTexte }
begin
 EditTexte.Text:= interpreteur.Filtrage(MemoSource.text);
end;

procedure TForm1.BitBtnExecClick(Sender: TObject);
{ OnClick du TBitBtn BitBtnExec "Exécuter" }
begin
 interpreteur.Lancer(EditTexte.Text);
end;

procedure TForm1.ButtonEffacerClick(Sender: TObject);
{ OnClick du TBitBtn BitBtnEffacer "Effacer" }
begin
 MemoSortie.Clear
end;

end.
```

Application console - seconde classe dérivée
Nous voulons maintenant écrire **une application console** utilisant la classe d'interpréteur TinterpretAbstr, selon le modèle ci-dessous :

```
D:\_coursinfo34\ChapProjets\Interpreteur\Console\InterpreteurLang.exe
Interpreteur du micro-langage
Vt = a, b,..,z, }, =, +, /, *, -, %, L, E, {, ;
Entrez un programme entre deux{ },termine par un "#"
Exemples:
---------
prog-1: {La;Lb;b=c;Ea;Eb;}#
prog-2: {La;}#
prog-3: {a=b+c;}#
prog-4: {Lb;Lc;a=c+b;d=a*c;Ea;Eb;Ec;Ed;}#
****************************************
{Lb;Lc;a=c+b;d=a*c;Ea;Eb;Ec;Ed;}#
****** INTERPRETE micro-langage ********
entrez b :100
entrez c :12
>> a = 112
>> b = 100
>> c = 12
>> d = 1344
```

Nous afficherons les résultats par une redéfinition de la méthode Ecrire à travers la procédure **writeln.**

Nous saisirons les valeurs par une redéfinition de la méthode Lire à travers la procédure **readln.**

Soit la nouvelle classe Tinterpreteur héritant de notre classe abstraite :

```
{ Tinterpreteur cas de Delphi en mode console }

Tinterpreteur=class(TinterpretAbstr)
   protected
    procedure Lire(symb:Vt;var x:integer); override;
    procedure Ecrire(symb:Vt;x:integer); override;
  end;
```

```
procedure Tinterpreteur.Ecrire(symb:Vt;x:integer);
begin
 writeln('>> ',symb,' = ', x)
end;

procedure Tinterpreteur.Lire(symb:Vt;var x: integer);
begin
 write('Entrez : ',symb,' :');
 readln(x)
 try
  readln(x);
 except
  x:=1;
 end
end;
```

A titre d'exercice simple, il vous est demandé d'écrire une unit Uinterprete contenant la classe Tinterpreteur précédente, puis le programme principal en Delphi mode console utilisant cette classe :

```
program InterpreteurLang;


{$APPTYPE CONSOLE}
uses sysutils , Uinterprete ;
var
  interpreteur:Tinterpreteur; { objet interpréteur }
begin
 interpreteur:=Tinterpreteur.Create(8);
  ....
 readln(mot);
 interpreteur.Lancer(mot);
end.
```

6.4 Mini-projet : réalisation d'un indentateur de code

Objectif : Utiliser les compétences aquises sur les structures de données, les tris et la notion d'automate à pile de mémoire pour construire un éditeur d'indentation de code du langage Delphi lui-même. Nous essayerons de dégager dans cet exemple, des éléments de construction qui pourront s'appliquer à d'autres langages classiques.

Soit le texte Delphi suivant (une implémentation de méthode) saisie par une personne :

```
implementation
procedure ClA1.Meth(var x:integer; y:real);
begin
if x < 5 then
y := y-7
else
while x in [a..b] do
begin
if expression(x,y)<'x' then
for x:=1 to 15 do
if expression(x,y)<'x' then
if expression(x,y)<'x' then
begin
x:=4;
end
else
if expression(x,y)<'x' then
begin
x:=4;
end
if expression(x,y)<'x' then
a:="bonjour"      //ceci est un commentaire
else   //ceci est un autre commentaire
wwww;
while dffg hh do
if dhdhdh then
dfdf
else
begin
b:="aaaaa"+"bbbbbb"+'c';
end
end;
end
end.
```

Soit le même texte saisie par une autre personne :

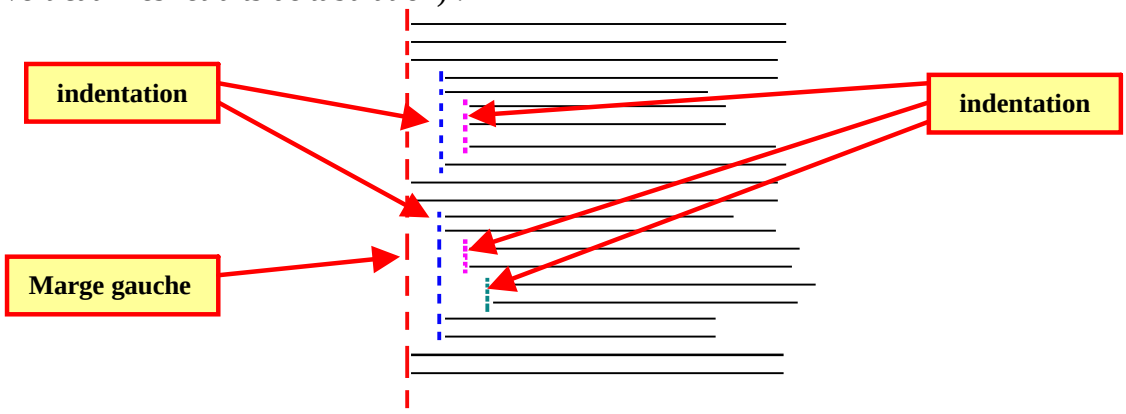
```
implementation
procedure ClA1.Meth(var x:integer; y:real);
begin if x < 5 then y := y-7 else
        while x in [a..b] do
    begin if expression(x,y)<'x' then for x:=1 to 15 do
  if expression(x,y)<'x' then
           if expression(x,y)<'x' then begin x:=4;
end else    if expression(x,y)<'x' then begin x:=4; end
   if expression(x,y)<'x' then  a:="bonjour"      //ceci est un commentaire
              else   //ceci est un autre commentaire
wwww; while dffg hh do
if dhdhdh then  dfdf else
           begin
                  b:="aaaaa"+"bbbbbb"+'c';
                      end end;
end; end.
```

Deux saisies différentes

Nous remarquons que nous sommes en face de deux saisies différentes du même texte et qu'aucune des deux ne fait apparaître la logique d'écriture par bloc du texte.
On souhaite alors construire un logiciel permettant une présentation du code fondée sur l'indentation (opération de mise en retrait de lignes selon des blocs logiques afin de présenter une meilleure lisibilité du code). La présentation restant un choix non figé, nous adopterons notre propre style d'indentation, le lecteur pourra se servir des outils fournis par la suite dans le logiciel pour changer certaines dispositions de présentation.

Une seule présentation

Nous adoptons le principe que les textes de code en général sont composés de mots qui sont soit des marqueurs syntaxiques, soit des marqueurs sémantiques, soit des marqueurs de séparation et des mots ordinaires. Les développeurs ont adopté comme présentation synthétique, la structure de "peigne" . Cette présentation est une combinaison de la "justification à gauche" d'un texte et de l'utilisation de tabulation dès qu"un bloc est ouvert ou fermé selon le schéma ci-dessous (les lignes horizontales figurent le texte, les pointillés verticaux les retraits de tabulation) :



Selon cette disposition, notre editeur de code devra pouvoir représenter les deux saisies précédentes du même texte de le même façon, soit comme ci-dessous (les mots clefs ont été surlignés en **noir gras** pour mieux faire ressortir l'indentation des lignes) :

```
implementation
procedure ClA1.Meth(var x:integer; y:real);
begin
   if x < 5 then
     y := y-7
   else
     while x in [a..b] do
     begin
       if expression(x,y)<'x' then
         for x:=1 to 15 do
           if expression(x,y)<'x' then
             if expression(x,y)<'x' then
             begin
               x:=4;
             end
             else
               if expression(x,y)<'x' then
```

```
            begin
              x:=4;
            end
        if expression(x,y)<'x' then
          a:="bonjour"      //ceci est un commentaire
        else   //ceci est un autre commentaire
          wwww;
        while dffg hh do
          if dhdhdh then
            dfdf
          else
          begin
            b:="aaaaa"+"bbbbbb"+'c';
          end
      end;
  end;
```

Spécifications de base du logiciel - Analyse et recherche des schémas d'indentation de base d'une unit Delphi

Identification des tâches pour la présentation d'un texte

Nous observons dans différents documents écrits manuellement ou par éditeur de code déjà existant qu'un certain nombre de règles non-écrites subordonnent la présentation de lignes de code. Nous remarquons que la structuration du langage influence d'une manière importante la présentation , en particulier pour les langages à structures de blocs où l'on pratique l'indentation (retrait d'une ligne par rapport à la précédente pour indiquer le début d'un nouveau bloc). Nous remarquons aussi qu'un certain nombre de styles de présentation sont laissés en libre choix.

Nous allons donc construire un logiciel pour langages structurés et plus particulièrement pour lignes de code Delphi, et nous adoptons l'idée que nos lignes de code doivent être le plus courtes possibles afin de ne pas surcharger leur lisibilité : nous priviligérons donc la briéveté par rapport au nombre plus important de lignes.

Nous postulons que la présentation du texte Delphi de 8 lignes suivant :

```
implementation
procedure ClA1.Meth(var x:integer; y:real);
begin if x < 5 then y := y-7 else     while x in [a..b] do
begin if expression(x,y)<'x' then for x:=1 to 15 do   if expression(x,y)<'x' then    if expression(x,y)<'x' then begin
x:=4;
end else    if expression(x,y)<'x' then begin x:=4; end
if expression(x,y)<'x' then  a:="bonjour"     //ceci est un commentaire
else   //ceci est un autre commentaire
end; end.
```

est "moins" lisible que le même texte beaucoup plus long (25 lignes) et réarrangé suivant (nous avons figuré avec des couleurs en gras les alignements de ligne):

```
implementation
```

```
procedure ClA1.Meth(var x:integer; y:real);
begin
  if x < 5 then
    y := y-7
  else
    while x in [a..b] do
    begin
      if expression(x,y)<'x' then
        for x:=1 to 15 do
          if expression(x,y)<'x' then
            if expression(x,y)<'x' then
            begin
              x:=4;
            end
            else
              if expression(x,y)<'x' then
              begin
                x:=4;
              end
      if expression(x,y)<'x' then
        a:="bonjour"     //ceci est un commentaire
      else   //ceci est un autre commentaire
    end;
end.
```

Principe adopté :

une ligne contient une instruction du langage.

Nous devons alors relever tous les schémas prévisibles de code et préciser le modèle de présentation que nous souhaitons leur voir suivre.

L'indentation d'une ligne

Nous repertorions ci-après les principales configurations de code possibles en partie gauche et leur comportement d'indentation souhaité en partie droite.

Imbrication de blocs

```
begin                          begin
begin                            begin
begin                              begin
begin                                begin
end                                  end
end                                end
end                              end
end                            end
```

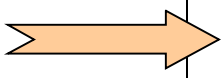
Lignes dans un bloc

```
begin                          begin
x:=58;                           x:=58;
y:='kkkkkk';                     y:='kkkkkk';
z:=-14.57                        z:=-14.57
end                            end
```

Instructions structurées

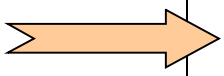
sans défaut de fermeture (repeat...until, try...except...end) :

```
begin
repeat
x:=58;
y:='kkkkkk';
z:=-14.57
until  (x> 76)and(g or not (x<7))  ;
end
```



```
begin
  repeat
    x:=58;
    y:='kkkkkk';
    z:=-14.57
  until  (x> 76)and(g or not (x<7))  ;
end
```

```
begin
try
x:=5
except
y:=x;
free;
end
end
```

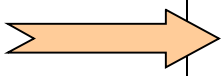


```
begin
  try
    x:=5
  except
    y:=x;
    free;
  end
end
```

Instructions d'itérations

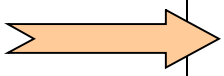
avec défaut de fermeture (for... , while....) :

```
begin
x:=26;
for i:=12 downto 5 do
x:=x+i;
y:=x-10;
end;
```



```
begin
  x:=26;
  for i:=12 downto 5 do
    x:=x+i;
  y:=x-10;
end;
```

```
begin
x:=26;
while(x<100) do
x:=x+i;
y:=x-10;
end;
```

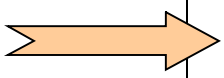


```
begin
  x:=26;
  while(x<100) do
    x:=x+i;
  y:=x-10;
end;
```

Instructions conditonnelles

avec défaut de fermeture (if...else) :

```
begin
x:=26;
if x<100 then
x:=x+i
else
x:=x-2
y:=x-10;
end;
```

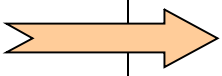


```
begin
  x:=26;
  if x<100 then
    x:=x+i
  else
    x:=x-2
  y:=x-10;
end;
```

Instructions conditionnelles imbriquées

(if...if...else...else) :

```
begin
if x<100 then
if x<200 then
if x<300 then
if x<400 then
if x<500 then
if x<600 then
x:=x+i
else
x:=x-2
y:=x-10;
end;
```

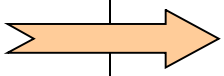


```
begin
  if x<100 then
    if x<200 then
      if x<300 then
        if x<400 then
          if x<500 then
            if x<600 then
              x:=x+i
            else
              x:=x-2
  y:=x-10;
end;
```

Mélange d'instructions à défaut de fermeture imbriquées

(rattachement du else pendant) :

```
begin
if x<100 then
if x<200 then
if x<300 then
if x<400 then
for x:=1 to 500 do
for x:=1 to 600 do
while x<700 do
if x<800 then
x:=x+i
else
x:=x-2
else
Autre(5);
y:=x-10;
end;
```

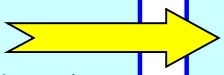


```
begin
  if x<100 then
    if x<200 then
      if x<300 then
        if x<400 then
          for x:=1 to 500 do
            for x:=1 to 600 do
              while x<700 do
                if x<800 then
                  x:=x+i
                else
                  x:=x-2
        else
          Autre(5);
  y:=x-10;
end;
```

Le découpage d'une ligne

Nous voulons que notre logiciel possède une certaine "intelligence" du texte et qu'il soit capable de découper automatiquement selon des modèles préétablis une ligne longue en plusieurs lignes. Par exemple, les 5 lignes de code de gauche devront se trouver transformées par l'éditeur de code en les 12 lignes de droite :

```
begin if x < 5 then y := y-7 else
        while x in [a..b] do
    begin if expression(x,y)<'x' then for x:=1 to 15
do
  if expression(x,y)<'x' then
          if expression(x,y)<'x' then begin x:=4;
```



```
begin
  if x < 5 then
    y := y-7
  else
    while x in [a..b] do
    begin
      if expression(x,y)<'x' then
        for x:=1 to 15 do
          if expression(x,y)<'x' then
            if expression(x,y)<'x' then
            begin
              x:=4:
```

Cette intelligence relative nécessite de fournir au logiciel des informations sur des **marqueurs sémantiques de découpage** d'une phrase. Prenons la phrase de code extraite du texte précédent :

```
begin if expression(x,y)<'x' then for x:=1 to 15 do
```

il est évident que le mot **begin** qui est déjà un mot clef du langage est aussi un marqueur sémantique de découpage, il en est de même pour les mots clefs **then** et **do** :

begin if expression(x,y)<'x' **then** for x:=1 to 15 **do**

Après la reconnaissance de ces marqueurs, le logiciel produira 3 lignes d'une seule instruction à partir de cette phrase :

begin
if expression(x,y)<'x' **then**
for x:=1 to 15 **do**

Spécifications opérationnelles du logiciel - Les grandes fonctions du logiciel et la méthodologie opératoire adoptée.

Mise en place de la présentation d'un texte

> Nous posons comme postulat que la présentation de notre code résultera de deux fonctions spécifiques du logiciel ceci afin de bien séparer les actions de **découpage** du texte et celle **d'indentation** proprement dite, nous rajouterons une troisième fonctionnalité plus tard : la coloration syntaxique des mots clefs du langage.

Méthode de découpage d'une ligne

Nous proposons de construire une méthode **CouperLigne** qui balaie entièrement toutes les lignes du texte dans un premier passage. A cette méthode **CouperLigne** nous attribuons deux fonctions :

Fonction découpage proprement dit, c'est la méthode **CouperLigne** qui effectuera la reconnaissance des marqueurs de découpage et qui réinserera immédiatement les nouvelles lignes générées, dans le texte comme dans l'exemple ci-dessous:

begin if x > 46 **then** for x:=46 downto 15 **do** y:=y+x **;** z:=y-1 ;

la méthode **CouperLigne** doit reconnaître les 4 marqueurs de découpage : **begin** , **then** , **do** et **;**

Après la reconnaissance de ces marqueurs la méthode **CouperLigne** doit insérer les 5 lignes ci-dessous dans le texte à la place de la ligne précédente :

```
begin
if x > 46 then
for x:=46 downto 15 do
y:=y+x ;
z:=y-1 ;
```

L'ensemble de tous les marqueurs de découpage est stocké dans une structure de données à accès direct afin d'être accessible immédiatement (un ensemble set of en Delphi), nous la noterons EnsMotDeCoupe.

Nous remarquons que la méthode **CouperLigne** doit pouvoir découper une ligne contenant un nombre quelconque et non connu à l'avance de marqueurs de découpage :

La méthode **CouperLigne** doit découper aussi bien la ligne :

begin if x > 46 **then** for x:=46 downto 15 **do** y:=y+x **;** z:=y-1 ;

que découper la ligne :

begin x:=5

Nous proposons de construire une méthode récursive qui va découper une ligne de gauche à droite en deux lignes, puis se rappeler récursivement sur la deuxième ligne jusqu'à épuisement des marqueurs de découpage. Prenons un exemple :

Soit la ligne de code suivante où pour des raisons de compréhension nous avons fait figurer son numéro dans la liste des lignes (ici ce serait la 17 ème ligne du texte) :

n° 17 - if x=0 then x:= 1 ; y:=5 begin

La méthode **CouperLigne** reconnaît le marqueur **then** elle scinde donc la ligne n°17 en 2 morceaux :

n° 17 - | if x=0 then | | x:= 1 ; y:=5 begin |

Puis la méthode **CouperLigne** remplace la ligne originale n°17 par deux nouvelles lignes n°17 et n°18 obtenues par découpage :

n° 17 - | if x=0 then | *n° 18* - | x:= 1 ; y:=5 begin |

Enfin la méthode **CouperLigne** se rappelle sur la ligne n°18 où elle reconnaît le marqueur **;** :

n° 18 - | x:= 1 ; | | y:=5 begin |

Elle réengendre 2 nouvelles lignes

n° 18 - x:= 1 ;
n° 19 - y:=5 begin

La méthode **CouperLigne** se rappelle une dernière fois récursivement sur la ligne n°19 et repère le marqueur **begin** , pour fournir finalement le nouveau texte suivant :

n° 17 - if x=0 then
n° 18 - x:= 1 ;
n° 19 - y:=5
n° 20 - begin

Traitement des commentaires

Nous savons par expérience que le code doit être documenté à l'aide de commentaires, l'attitude des développeurs est d'utiliser soit un commentaire mono-ligne pour une information brève et ciblée sur une ligne de code particulière, soit un commentaire plus long de plusieurs lignes que nous dénommerons commentaire multi-lignes. La méthode **CouperLigne** doit respecter les commentaires en particulier les commentaires mono-ligne.

1)En effet la ligne de code Delphi suivante :

n° 17 - if x=0 then // ceci est un commentaire

ne doit pas être découpée en les 2 lignes suivantes :
n° 17 - if x=0 then
n° 18 -// ceci est un commentaire

2)En revanche la ligne :

n° 17 - if x=0 then x:= 1 ; y:=5 // ceci est un commentaire

sera découpée en :
n° 17 - if x=0 then
n° 18 - x:= 1 ;
n° 19 - y:=5 // ceci est un commentaire

Nous normalisons la présentation des commentaires multi-lignes afin d'avoir une représentation standard dans tous les textes,en Delphi :

```
{  Les nombres avec un séparateur décimal
   ou un exposant désignent des réels, alors que
   les autres nombres désignent des entiers.
}
```

Entre un marqueur syntaxique de début de commentaire multi-lignes ('**{**'en Delphi) et un marqueur de fin de commentaire multi-lignes ('**}**'en Delphi), les lignes de commentaires ne subiront pas de présentation particulière, et elles ne seront ni découpées, ni indentées.

Remarque importante :

Les mots d'une ligne de code peuvent être différenciés selon leur mode d'interprétation soit rien (aucune action, c'est alors un mot banal), soit c'est un marqueur syntaxique (début de bloc, fin de bloc,...), soit c'est un marqueur sématique de découpage, soit c'est un mot clef (utilisable pour la coloration syntaxique des mots clefs). Certains éléments peuvent être à la fois marqueur syntaxique , marqueur de découpage etc.. (comme par exemple le mot clef begin). Nous englobons toutes ces catégories sous un seul vocable: les catégories lexicales que nous dénoterons **CategLexeme**.

Voici ci-dessous les catégories lexicales **CategLexeme** qui ont été utilisées dans le logiciel construit :

isRien, isStartBloc, isEndBloc ,isParagraphe, isAlinea, isTeteBloc, isMilieuInstrStruct, isDebutDefautFermeture, isFinDeLigne, isMilieuDefautFermeture, isDebutDefautFermetureCompos, isDebutInstrStruct, isEndInstrStruct, IsStartComment, IsEndComment, IsComment, isChaine, isDebutParagrapheTry, isMilieuParagrapheTry, isAsLignePrec, isDelimit, isVide .

Nous construirons une méthode qui permet d'extraire les mots d'une ligne de code et une méthode qui fournit la catégorie lexical d'un mot, nous aurons ainsi à notre disposition un petit analyseur lexical pour l'indentation (et le coloriage)

Stockage des information collectées

Nous assignons aussi à la méthode **CouperLigne** un travail de conservation dans une liste des informations collectées sur une ligne. Sachant que la prochaine activité d'indentation du logiciel necessite pour une ligne donnée de connaître le genre de ligne de la précédente nous stockons dans une liste **ListeFirstKeyLine** un lexème (la catégorie lexicale du premier mot de la ligne ainsi que le mot lui-même).

Ainsi, en reprenant l'exemple précédent après action de la méthode **CouperLigne** sur la ligne
n° 17 - if x=0 then x:= 1 ; y:=5 // ceci est un commentaire

on obtient un nouveau texte :

....
n° 17 - **if** x=0 then
n° 18 - **x** := 1 ;
n° 19 - **y** :=5
n° 20 - **begin**

et une liste de lexèmes **ListeFirstKeyLine** qui s'agrandit de 4 éléments :

.....
n° 17 - (**if** , isDebutDefautFermetureCompos)
n° 18 - (**x** , isRien)
n° 19 - (**y** , isRien)
n° 20 - (**begin** , isStartBloc)

Moteur de décisions

Nous utilisons pour prendre les décisions de présentation d'une ligne de code, un moteur de décision fondé sur **un automate à pile de mémoire**.

L'analyse et la construction se font ligne à ligne les décisions d'actions sont dirigées par un automate à pile qui permet pour chaque ligne de savoir dans quel contexte la ligne se situe par connaissancede la catégorie du premier lexème de la ligne (état de l'automate) et par consultation d'une pile contextuelle dénotée **PileBlocs,** contenant des informations sur les blocs imbriqués et sur les instructions à défaut de fermeture imbriquées dans les blocs.

Nous rappellons que certains langages comme Delphi, Java, C++, C# etc... possèdent des instructions structurées avec un défaut de fermeture (pas de marqueur syntaxique de fin de l'instruction), Visual Basic quant à lui ne possède pas d'instruction à défaut de fermeture.

Par exemple l'instruction de boucle **while** fermée en VB et non fermée dans les autres langages Delphi, Java, C++, C# :

En Delphi :
while x < 5 **do**
x := x+2
	En Delphi :
	while x < 5 **do**
	begin
	x := x+2;
	y:=x-1;
	end

En C++, Java, C# :
while (w < 5)
x +=2;
	En C++, Java, C# :
	while (w < 5)
	{
	x += 2;
	y = x-1;
	}

En VB : **while** x < 5 x = x+2 **wend**	En VB : **while** x < 5 x = x+2 y = x-1 **wend**

Notre moteur de décisions doit donc être capable dans un langage comme Delphi de proposer une indentation particulière pour de telles instructions. En reprenant l'exemple de l'instruction **while**, nous aurons :

texte de la ligne avant indentation	texte de la ligne après indentation

while x < 5 **do**
begin
x := x+2;
y:=x-1;
end
	while x < 5 **do**
	begin
	  x := x+2;
	  y:=x-1;
	end

while x < 5 **do**
x := x+2;
y:=x-1;
	while x < 5 **do**
	  x := x+2;
	y:=x-1;

Dans le cas où plusieurs instructions de ce type sont imbriquées, le moteur de décision doit être capable de situer correctement la dernière ligne qui "ferme" les imbrications et de rattacher la prochaine instruction au bon bloc.

Exemple :

texte de la ligne avant indentation	texte de la ligne après indentation

for i:=1 to 10 **do**
while x < 7 **do**
while x < 6 **do**
while x < 5 **do**
begin
x := x+2;
y:=x-1;
end
	for i:=1 to 10 **do**
	  while x < 7 **do**
	  while x < 6 **do**
	  while x < 5 **do**
	  begin
	  x := x+2;
	  y:=x-1;
	  end

for i:=1 to 10 **do**
while x < 7 **do**
while x < 6 **do**
while x < 5 **do**
x := x+2;
y:=x-1;
	for i:=1 to 10 **do**
	  while x < 7 **do**
	  while x < 6 **do**
	  while x < 5 **do**
	  x := x+2;
	y :=x-1;

Pour ce faire l'automate de décision travaille avec la pile contextuelle **PileBlocs** dans laquelle il range les ouvertures de blocs structurés et les rangs d'indentation à chaque fois qu'il rencontre une instruction à défaut de fermeture imbriquée (lexèmes : isDebutDefautFermeture et isDebutDefautFermetureCompos).

Enfin, dans la catégorie des instructions à défaut de fermeture, le **if...else** prend une place particulière due au problème de la résolution classique du **else** pendant (nous attribuons au **if** une categorie lexicale spéciale isDebutDefautFermetureCompos et au **else** la catégorie isMilieuDefautFermeture) :

if P1 **then if** P2 **then else** A **else** B

Problème d'ambiguïté résolue classiquement par le compilateur par rattachement du premier 'else' au 'if' le plus proche. Notre logiciel doit tenir compte de cette démarche et fournir une présentation adéquate qui prenne en compte ce rattachement au if le plus proche :

if P1 **then if** P2 **then else** A **else** B ⟹

```
if P1 then
   if P2 then
   else A
else B
```

Notre logiciel doit pouvoir interpréter et indenter aisément des situations comme celle qui suit (les couleurs ne sont mises que pour bien faire ressortir les alignements des lignes) :

texte de la ligne avant indentation	texte de la ligne après indentation

for i:=1 to 10 **do**
if y<4 **then**
if y<3 **then**
while x < 7 **do**
if y<2 **then**
while x < 6 **do**
while x < 5 **do**
x := x+2
else
y := x-1
else
z := 8 ;
	for i:=1 to 10 **do**
	if y<4 **then**
	if y<3 **then**
	while x < 7 **do**
	if y<2 **then**
	while x < 6 **do**
	while x < 5 **do**
	x := x+2
	else
	y := x-1
	else
	z := 8 ;

Les piles contextuelles

Nous utilisons une pile auxiliaire de contexte que nous notons **PileIndent,** qui contient uniquement les valeurs des retraits de chaque début de bloc. L'ensemble des deux piles permet à l'automate d'avoir une **information constante** sur la valeur du retrait à attribuer à une ligne à l'intérieur d'un bloc. Les piles **PileIndent** et **PileBlocs** sont liées logiquement entre elles selon le schéma d'information ci-dessous :

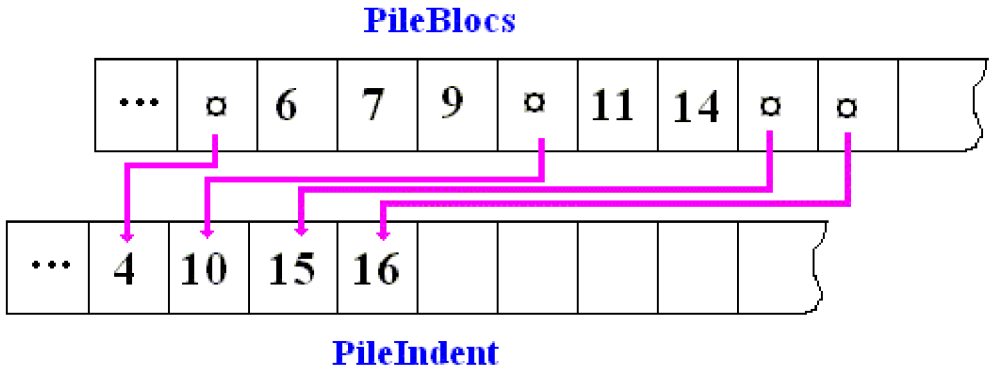
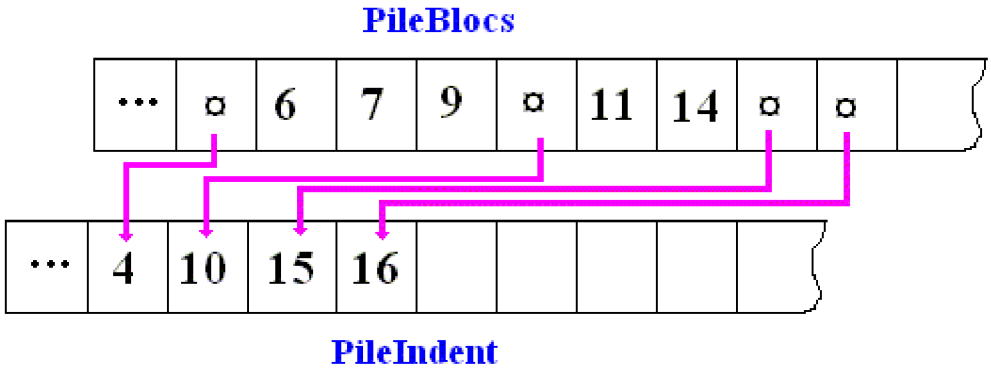


Schéma de piles pouvant correspondre aux lignes de code suivantes :

lignes abstraites	lignes en Delphi
< ... , isStartBloc > < ... , isDebutDefautFermetureCompos > < ... , isDebutDefautFermetureCompos > < ... , isDebutDefautFermeture > < ... , isDebutDefautFermetureCompos > < ... , isStartBloc > < ... , isDebutDefautFermetureCompos > < ... , isDebutDefautFermeture > < ... , isDebutDefautFermeture > < ... , isDebutDefautFermetureCompos > < ... , isStartBloc > < ... , isRien > < ... , isStartBloc >	Retrait=4 / begin Retrait =6 / if .... then Retrait =7 / if .... then Retrait =8 / for ... to Retrait =9 / if .... then Retrait =10 / begin Retrait =11 / if .... then Retrait =12 / for ... to Retrait =13 / while ... do Retrait =14 / if .... then Retrait =15 / begin Retrait =16 / x := 47 Retrait =16 / begin



Empilement :

> Lorsqu'une marque de début de bloc ⊗ est stockée dans la **PileBlocs** , la valeur actuelle de la position du retrait d'indentation est stockée dans la pile auxiliaire **PileIndent**.

Dépilement :

> A chaque fois qu'une marque de début de bloc ⊗ est dépilée de la pile **PileBlocs** , la pile **PileIndent** est dépilée parallèlement de son sommet de telle façon que le sommet de **PileIndent** représente toujours le niveau d'indentation du bloc en cours.

Ainsi les deux piles **PileBlocs** et **PileIndent** représentent une vue abstraite de tout le texte sous l'angle de la présentation. Elles sont utilisées par un automate de décision qui les remplit, les consulte et les modifie en fonction de situations spécifiques.

Méthodes de base

Il nous faut donc construire nos outils de base permettant ces différents types de positionnement. Nous listons ci-dessous les méthodes de notre classe éditeur qui travaillent sur une ligne de texte et que nous avons classées pour des raisons pédagogiques, par catégories :

```
{ Méthodes de positionnement d'une ou plusieurs lignes }
function  GetPosIndent(NumLigne:integer):integer;
function FirstChar(NumLigne:integer):integer;
function NbrBlancs(NumLigne:integer):integer;
function Decalage(nbr:integer):string;
function DecalageTAB(nbrTab:integer):string;
procedure AjusteLigneNextTAB(NumLigne:integer);
procedure AjusteLigneRight(NumLigne:integer);
procedure AvancerLigne(NumLigne,Nbrtab:integer);
procedure ReculerLigne(NumLigne,Nbrtab:integer);
procedure PositionneSurLignePrec(NumLigne:integer);
procedure PositionneLigneSurIndent(NumLigne:integer;Depiler:boolean);
procedure PositionneLigneDeFinDefautFermerure(NumLigne:integer);
procedure PositionneLigneSurLastDefautFermeture(NumLigne:integer);
procedure AlignerPargrf(Debut,Fin:integer;tabuler:boolean);
procedure AvancerPargrf(Debut,Fin:integer);
procedure ReculerPargrf(Debut,Fin:integer);

{ Gestion des piles }
procedure RAZPileIndent;
procedure RAZPileBlocs;
procedure DepileNextMarkBloc;

{ Analyseur lexical }
procedure ExtraitMot(Ligne:String; var numcar:integer; var Mot:string;  var IsMot: MotsLimites;var
TypeBloc:CategLexeme);

{ Découpage d'une ligne }
procedure StockKeyLine(NumLigne:integer);
procedure CouperLigne(NumLigne:integer);

{ Indentation d'une ligne : Moteur de décision }
procedure IndentLigneIfMotNotRien(NumLigne:integer;Motactuel,Motprec:string;EtatMot,EtatmotPrec:
CategLexeme);
procedure IndentLigneIfMotisRien(NumLigne:integer;Motactuel,Motprec:string;EtatMot,EtatmotPrec:
CategLexeme);
procedure IndentUneLigne(NumLigne:integer;indentTout:boolean);

{ Analyse et indentation de tout le texte }
procedure IndenterTout;
procedure AnalyseComplete;
```

Toutes ces méthodes participent aux actions de découpage et d'indentation proprement dite de l'automate. Nous décrivons ci-après les deux actions.

Découpe d'une ligne

Eléments utiles au découpage d'une ligne :

- Méthode ExtraitMot (Ligne, rang, Mot, MotIs, MotIsBloc) est l'analyseur lexical qui extrait consécutivement dans une ligne les mots (dans la variable Mot) et leur catégorie lexicale (dans la variable MotIsBloc).
- Méthode StockKeyLine(NumLigne:integer) range le premier mot de la ligne et sa catégorie lexicale dans une liste nommée ListeFirstKeyLine.
- Rappelons enfin que toutes les catégories lexicales sont décrites dans le type **CategLexeme** :

> isRien, isStartBloc, isEndBloc ,isParagraphe, isAlinea, isTeteBloc, isMilieuInstrStruct, isDebutDefautFermeture, isFinDeLigne, isMilieuDefautFermeture, isDebutDefautFermetureCompos, isDebutInstrStruct, isEndInstrStruct, IsStartComment, IsEndComment, IsComment, isChaine, isDebutParagrapheTry, isMilieuParagrapheTry, isAsLignePrec, isDelimit, isVide .

Nous ne rentrerons pas dans le détail du code, seulement dans les grandes lignes structurelles afin de comprendre comment l'automate prend ses décisions voici en texte algorithmique :

```
Algorithme CouperLigne
entrée : NumLigne ∈ entier
...
EnsMotdeCoupe = { isFinDeLigne, isMilieuInstrStruct, isStartBloc, isEndInstrStruct,
isMilieuDefautFermeture, isEndBloc, isTeteBloc, isStartComment, isEndComment, isComment }

début
 Tantque ( non fin de la ligne ) et ( MotIsBloc ∉ EnsMotdeCoupe ) faire
    ...
    ExtraitMot ( Ligne, rang, Mot, MotIs, MotIsBloc );
    ...
    si MotIsBloc ∈ {IsComment, isMilieuParagrapheTry, isTeteBloc}alors
      StockKeyLine ( NumLigne );
      sortir de la boucle;
   fsi ;
    si MotIsBloc ∈ {isStartBloc, isMilieuInstrStruct, isMilieuDefautFermeture,
           isEndInstrStruct, isEndBloc, IsStartComment, IsEndComment} alors
      Traiter les cas de découpage de la ligne et des commentaires
    sinon
    si (MotIsBloc ∈ {isFinDeLigne} ) et (non ChaineEnCours) alors
     ligne à décomposer et ajouter les nouvelles lignes dans le texte
    fsi
  fsi
 fTant
 StockKeyLine ( NumLigne );
fin-CouperLigne
```

Exemple :

Afin de bien comprendre ce que fait ce premier traitement sur le texte prenons le texte Delphi de 6 lignes suivant :

```
for i:=1 to 10 do if y<4 then
begin  if y<3 then while x < 7 do begin
if y<2 then  while x < 6 do while x < 5 do
x := x+2 else y := x-1
end end
else z := 8 ;
```

Appliquons 6 fois (une fois pour chaque ligne) la méthode CouperLigne (en rouge souligné le premier marqueur de découpage repéré par la méthode)

Entrée de la méthode **CouperLigne** : une ligne de code.	sortie de **CouperLigne** : une ou plusieurs lignes.	Ajout dans la liste ListeFirstKeyLine
for i:=1 to 10 **do** **if** y<4 **then**	**for** i:=1 to 10 **do** **if** y<4 **then**	< Mot = **for** , MotIsBloc >

if y<2 **then** **while** x < 6 **do while** x < 5 **do**
 	
La deuxième ligne est sécable :
while x < 6 **do while** x < 5 **do**
	if y<2 **then**
	while x < 6 **do while** x < 5 **do**
	
	par rappel récursif :
	while x < 6 **do**
	while x < 5 **do**
		< Mot = **if** , MotIsBloc >

etc...

Rappelons que MotIsBloc ∈ **CategLexeme**, voici les résultats effectifs produits :

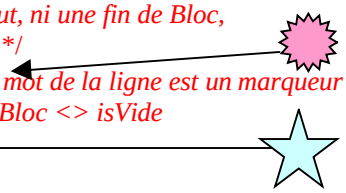
Texte après découpage	liste ListeFirstKeyLine

for i:=1 to 10 **do**
if y<4 **then**
begin
if y<3 **then**
while x < 7 **do**
begin
if y<2 **then**
while x < 6 **do**
while x < 5 **do**
x := x+2
else
y := x-1
end
end
else
z := 8 ;
	< **for** , isDebutDefautFermeture >
	< **if** , isDebutDefautFermetureCompos >
	< **begin** , isStartBloc >
	< **if** , isDebutDefautFermetureCompos >
	< **while** , isDebutDefautFermeture >
	< **begin** , isStartBloc >
	< **if** , isDebutDefautFermetureCompos >
	< **while** , isDebutDefautFermeture >
	< **while** , isDebutDefautFermeture >
	< x , isRien >
	< **else** , isMilieuDefautFermeture >
	< y , isRien >
	< **end** , isEndBloc >
	< **end** , isEndBloc >
	< **else** , isMilieuDefautFermeture >
	< z , isRien >

Algorithmes d'indentation

L'indentation est effectuée par un nouveau passage sur le texte, pour des raisons de clarté les actions ont été divisées en 3 algorithmes interdépendants.

```
Algorithme IndentUneLigne
entrée : NumLigne ∈ entier
.........
Mot : premier mot de la ligne
MotIsBloc : catégorie lexicale du premier mot de la ligne

début
    ...
 si MotIsBloc = IsStartComment alors
   ...// c'est un début de commentaire multi-lignes
 sinon
   si MotIsBloc = IsEndComment alors
      ...//le découpage de ligne met une fin de commentMulti sur une ligne seule
   sinon
     si MotIsBloc = IsChaine alors
       ...// le début de ligne est une chaine
     sinon
       si MultiEnCours alors
         ... //on est dans un commentaire multi-lignes
       fsi
     fsi
   fsi
 fsi
  ....
 si MotIsBloc ∈ [{ isRien , isVide } alors
 /* le 1er mot de la ligne n'est ni un début, ni une fin de Bloc,
  ni un paragraphe , ou la ligne est vide */
   IndentLigneIfMotisRien // le 1er mot de la ligne est un marqueur d'indentation
 sinon // MotIsBloc <> isRien ou MotIsBloc <> isVide
   IndentLigneIfMotNotRien
 fsi
fin-IndentUneLigne
```



```
Algorithme  IndentLigneIfMotNotRien
 // examine l'état lexical du premier mot de la ligne actuelle pour décider
   on examine tous les cas sauf isRien et isVide
début
  selon la valeur de EtatMot de la ligne actuelle
   IsComment, IsStartComment :
      selon la valeur de EtatmotPrec de la ligne précédente
       isStartBloc,isDebutInstrStruct,isMilieuInstrStruct,
       isMilieuDefautFermeture,isDebutDefautFermeture,
       isDebutDefautFermetureCompos :
           Avancer la ligne d'une tabulation

       isEndBloc, isEndInstrStruct, isRien :
          si  on est dans un bloc à défaut de fermeture alors
            on Positionne la Ligne sur la valeur De Fin de DefautFermerure
```

```
        sinon
          on Positionne  la Ligne Sur la  Ligne Précédente
        fsi

      tous les autres cas :
        on Positionne  la Ligne Sur la  Ligne Précédente

    fin selon la valeur de EtatmotPrec

isStartBloc, isDebutInstrStruct :
    si EtatmotPrec  ∈  { isStartBloc, isDebutInstrStruct,
              isDebutParagrapheTry, isMilieuParagrapheTry, isMilieuInstrStruct } alors
       Avancer la ligne d'une tabulation
    sinon
      si EtatmotPrec  ∈  { isAsLignePrec, IsEndComment, IsComment, IsStartComment } alors
        on Positionne  la Ligne Sur la  Ligne Précédente
      sinon
        Traitement des cas des blocs à defaut fermeture
      fsi
    fsi
    Empiler dans PileBlocs une marque de début de bloc ;
    Empiler dans PileIndent la valeur de la position d'indentation actuelle

isEndBloc, isEndInstrStruct :
   on dépile PileBlocs jusqu'à la prochaine marque de début de bloc ;
   on Positionne la ligne Sur l'Indent ation actuelle ;
   on examine le sommet de PileBlocs qui sert
   à positionner un drapeau de déclaration ou de fin de bloc à défaut de fermeture

isParagraphe :
   traitement du cas du paragraphe ;

isDebutParagrapheTry:
  traitement du cas du paragraphe du type try ;

isMilieuParagrapheTry:
  on Positionne la ligne Sur l'Indent ation actuelle ;

isAlinea:
  traitement du cas de l'alinéa ;

isTeteBloc:
  traitement du cas dela tête de bloc ;

isMilieuInstrStruct :
  traitement du cas d'un milieu d'instruction structurée ;

isDebutDefautFermeture, isDebutDefautFermetureCompos :
  traitement du cas des débuts de bloc à défaut de fermure;

isMilieuDefautFermeture :
  si  on est dans un bloc à défaut de fermeture alors
        on Positionne la Ligne sur la dernière indentation d'ouverture de  Defaut Fermerure
   sinon
        on Positionne  la Ligne Sur la  Ligne Précédente
  fsi

isAsLignePrec :
  on Positionne  la Ligne Sur la  Ligne Précédente
```

```
 fin selon la valeur de EtatMot
fin-IndentLigneIfMotNotRien
```

```
Algorithme  IndentLigneIfMotisRien
 // l'état lexical du premier mot de la ligne actuelle est isRien ou isVide
   on examinel'état lexical du premier mot de la ligne précédente
début
   selon la valeur de EtatmotPrec de la ligne précédente
       isStartBloc, isDebutInstrStruct, isParagraphe,
        isAlinea, isTeteBloc, isMilieuInstrStruct, isMilieuParagrapheTry :
           Avancer la ligne d'une tabulation

    IsStartComment, isVide, IsEndComment, IsComment :
          on Positionne  la Ligne Sur la  Ligne Précédente

    isDebutDefautFermeture, isDebutDefautFermetureCompos,  isMilieuDefautFermeture :
          Avancer la ligne d'une tabulation

    isRien,isEndBloc, isEndInstrStruct :
      si  on est dans un bloc à défaut de fermeture alors
            on Positionne la Ligne sur la valeur De Fin de DefautFermerure
      sinon
            on Positionne  la Ligne Sur la  Ligne Précédente
      fsi
  fin selon la valeur de EtatmotPrec
fin-IndentLigneIfMotisRien
```

Les piles contextuelles

Après avoir produit et **normalisé** un nouveau texte et la liste des premiers lexèmes, l'automate a repris le nouveau texte **normalisé** pour l'indenter à l'aide des algorithmes précédents.

Nous avons juste mentionné la gestion des piles qui est essentielle aux bonnes prises de décision. Afin de cerner de plus près l'intérêt de ces piles , nous reprenons l'exemple donné au paragraphe des spécifications opérationnelles, puis nous le modifierons :

texte de la ligne avant indentation	texte de la ligne après indentation

for i:=1 to 10 **do**
if y<4 **then**
if y<3 **then**
while x < 7 **do**
if y<2 **then**
while x < 6 **do**
while x < 5 **do**
x := x+2
else
y := x-1
else
z := 8 ;
	for i:=1 to 10 **do**
	if y<4 **then**
	if y<3 **then**
	while x < 7 **do**
	if y<2 **then**
	while x < 6 **do**
	while x < 5 **do**
	x := x+2
	else
	y := x-1
	else
	z := 8 ;

La gestion de la pile contextuelle **PileBlocs** est alors essentielle dans ce genre d'éventualité : l'automate empile la valeur actuelle de retrait de chaque lexème du type isDebutDefautFermetureCompos, l'automate dépile à chaque lexème de catégorie isMilieuDefautFermeture.

La pile **PileBlocs** contient uniquement des symboles de **marquage de début de bloc** ⊗ et de **rang d'indentation d'une ligne de catégorie à défaut fermeture** :

Reprenons l'exemple précédent et visualisons l'évolution prévue de la pile **PileBlocs** :

for i:=1 to 10 **do**	[.....] rien n'est empilé (retrait actuel=1)
if y<4 **th**en	[....., **2**] retrait actuel=2 empilé
if y<3 **th**en	[....., 2 , **3**] retrait actuel=3 empilé
while x < 7 **do**	[....., 2 , 3] rien n'est empilé (retrait actuel=4)
if y<2 **then**	[....., 2 , 3 , **5**] retrait actuel=5 empilé
while x < 6 **do**	[....., 2 , 3 , 5] rien n'est empilé (retrait actuel=6)
while x < 5 **do**	[....., 2 , 3 , 5] rien n'est empilé (retrait actuel=7)
x := x+2	[....., 2 , 3 , 5] rien n'est empilé (retrait actuel=7) décalage +1 par rapport à la ligne précedente
else	[....., 2 , 3] **5** est dépilé (retrait actuel=5)
y := x-1	[....., 2 , 3] rien n'est empilé (retrait actuel=5) décalage +1 par rapport à la ligne précedente
else	[....., 2] **3** est dépilé (retrait actuel=3)
z := 8 ;	[....., 2] rien n'est empilé (retrait actuel=3) décalage +1 par rapport à la ligne précedente

Si la prochaine ligne est un **else** (isMilieuDefautFermeture) le sommet de pile (valeur=2) indique le bon retrait, sinon la pile contextuelle **PileBlocs** n'est pas consultée et c'est une autre procédure de décision qui est mise en oeuvre. Nous n'avons pas fait figurer la pile **PileIndent**, car celle-ci n'a pas évolué dans ce morceau de code à indenter du fait qu'aucun nouveau bloc n'a été ouvert. **PileIndent** contient dès le départ la valeur 0, soit la valeur du premier retrait du texte.

Introduisons des blocs dans ce texte :

texte de la ligne avant indentation	texte de la ligne après indentation

for i:=1 to 10 **do**
if y<4 **then**
begin
if y<3 **then**
	for i:=1 to 10 **do**
	  if y<4 **then**
	  begin
	  if y<3 **then**

```
while x < 7 do
begin
if y<2 then
while x < 6 do
while x < 5 do
x := x+2
else
y := x-1
end
end
else
z := 8 ;
```

```
while x < 7 do
begin
 if y<2 then
   while x < 6 do
     while x < 5 do
       x := x+2
 else
   y := x-1
end
end
else
  z := 8 ;
```

Visualisons sous la forme de deux tableaux, la nouvelle évolution prévue de la pile **PileBlocs** (⊗ = marque de début de bloc) et celle de la pile **PileIndent** :

for i:=1 to 10 **do**	**PileBlocs** = [.....] rien n'est empilé (retrait actuel=1) **PileIndent** = [.....]
if y<4 **then**	**PileBlocs** = [....., **2**] 2 empilé retrait actuel=2 **PileIndent** = [.....]
begin	**PileBlocs** = [....., 2 , ⊗] marque de bloc empilée (retrait actuel=2) **PileIndent** = [..... , 2]
if y<3 **then**	**PileBlocs** = [....., 2 , ⊗ , 3] 3 empilé retrait actuel=3 **PileIndent** = [..... , 2]
while x < 7 **do**	**PileBlocs** = [....., 2 , ⊗ , 3] rien n'est empilé (retrait actuel = **4**) **PileIndent** = [..... , 2]
begin	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗] marque de bloc empilée (retrait actuel = **4**) **PileIndent** = [..... , 2 , 4]
if y<2 **then**	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗ , **5**] 5 empilé retrait actuel=5 **PileIndent** = [..... , 2 , 4]
while x < 6 **do**	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗ , 5] rien n'est empilé (retrait actuel=6) **PileIndent** = [..... , 2 , 4]
while x < 5 **do**	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗ , 5] rien n'est empilé (retrait actuel=7) **PileIndent** = [..... , 2 , 4]
x := x+2	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗ , 5] rien n'est empilé (retrait actuel=8) décalage +1 par rapport à la ligne précedente

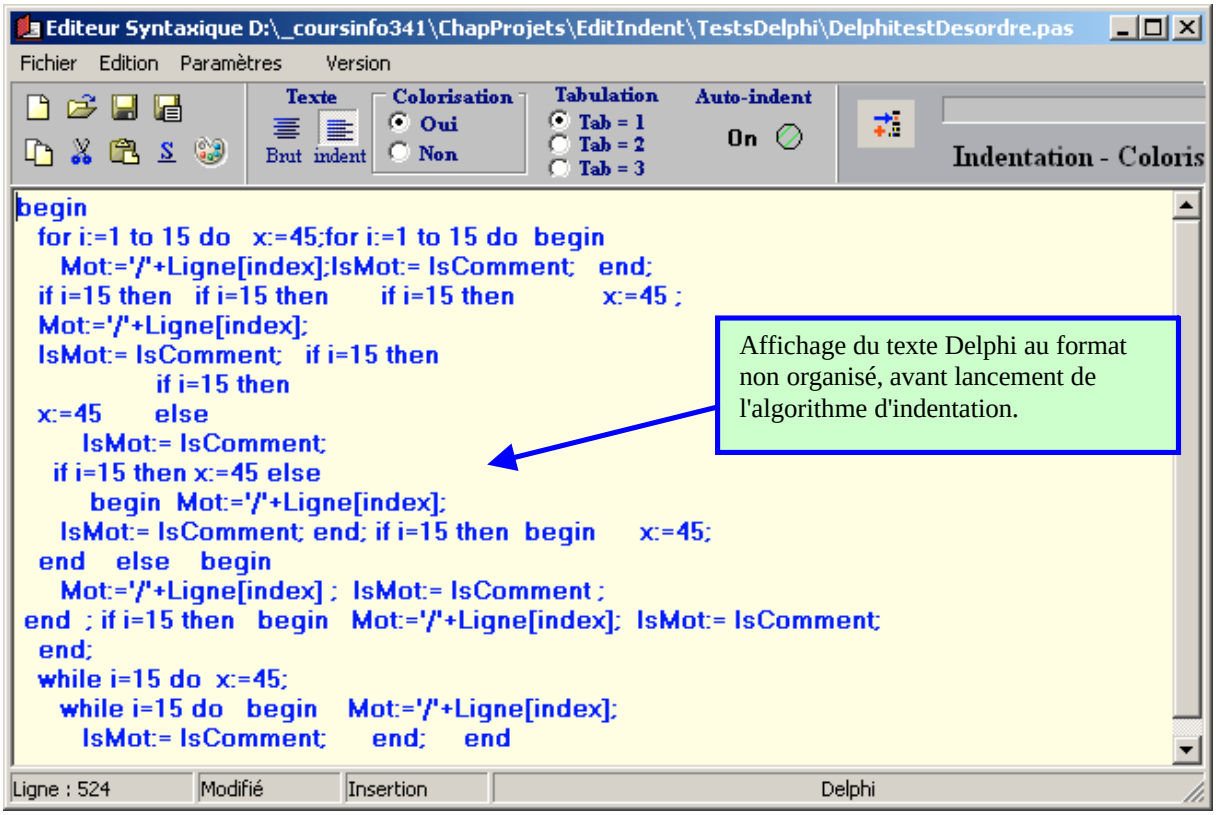
	PileIndent = [..... , 2 , 4]
else	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗] **5** est dépilé (retrait actuel=5) **PileIndent** = [..... , 2 , 4]
y := x-1	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗] rien n'est empilé (retrait actuel=6) décalage +1 par rapport à la ligne précedente **PileIndent** = [..... , 2 , 4]
end	**PileBlocs** = [....., 2 , ⊗ , 3] ⊗ est dépilé (retrait actuel = 4) **PileIndent** = [..... , 2] , 4 est dépilé
end	**PileBlocs** = [....., 2] **⊗ , 3 sont dépilés** (retrait actuel=2) **PileIndent** = [.....] , 2 est dépilé
else	**PileBlocs** = [.....] **2** est dépilé (retrait actuel=2)
z := 8 ;	**PileBlocs** = [.....] rien n'est empilé (retrait actuel=3)

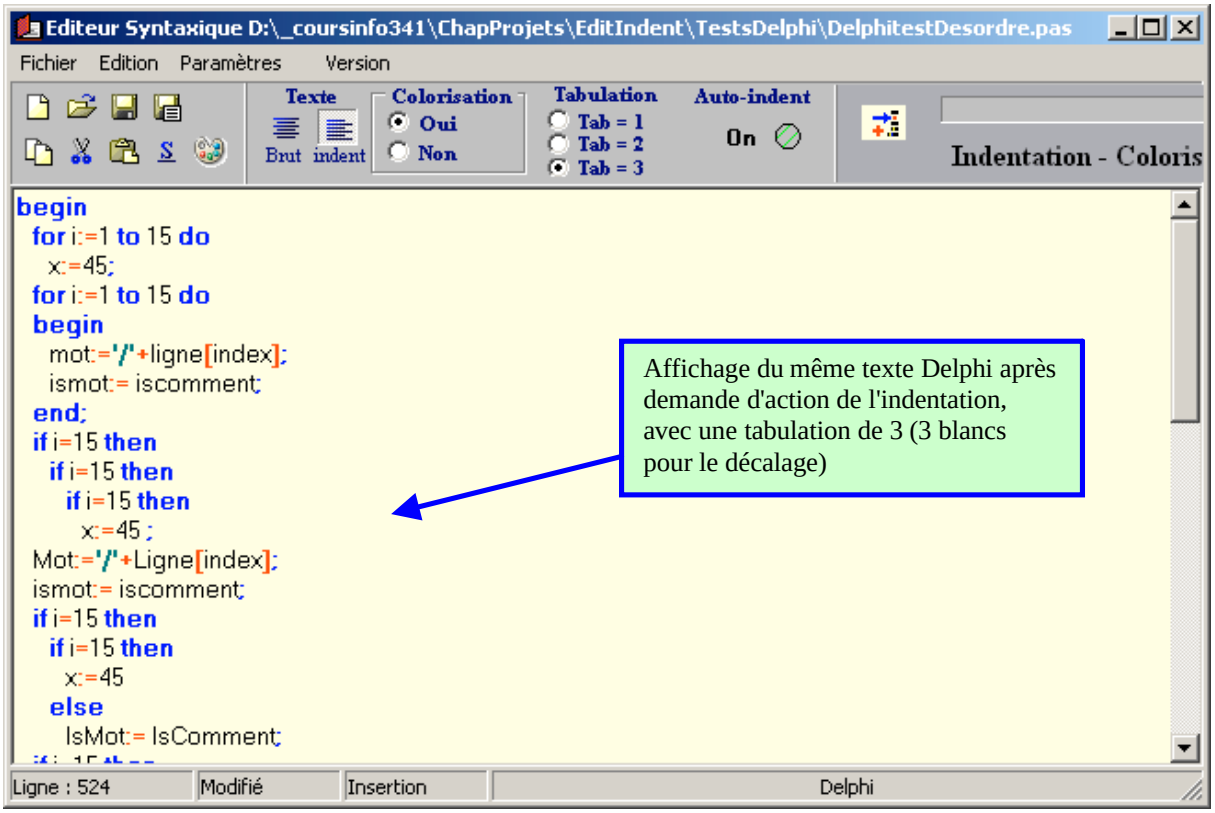
[retrait] = Valeur du curseur d'indentation	**PileBlocs**	**PileIndent**
[1] **for** i:=1 to 10 **do** [2] **if** y<4 **then** [2] **begin** [3] **if** y<3 **then** [4] **while** x < 7 **do** [4] **begin** [5] **if** y<2 **then** [6] **while** x < 6 **do** [7] **while** x < 5 **do** [8] x := x+2 [5] **else** [6] y := x-1 [4] **end** [2] **end** [2] **else** [3] z := 8 ;	[.....] [..... , 2] [....., 2 , ⊗] [....., 2 , ⊗ , 3] [....., 2 , ⊗ , 3] [....., 2 , ⊗ , 3 , ⊗] [....., 2 , ⊗ , 3 , ⊗ , 5] [....., 2 , ⊗ , 3 , ⊗ , 5] [....., 2 , ⊗ , 3 , ⊗ , 5] [....., 2 , ⊗ , 3 , ⊗ , 5] [....., 2 , ⊗ , 3 , ⊗] [....., 2 , ⊗ , 3 , ⊗] [....., 2 , ⊗ , 3] [....., 2] [.....] [.....]	[.....] [.....] [..... , 2] [..... , 2] [..... , 2] [..... , 2 , 4] [..... , 2 , 4] [..... , 2 , 4] [..... , 2 , 4] [..... , 2 , 4] [..... , 2 , 4] [..... , 2 , 4] [..... , 2] [.....] [.....] [.....]

Nous voyons que les deux piles permettent à l'automate de connaître en permanence en **valeur absolue de tabulations**, la profondeur de retrait du bloc où se trouve la ligne à indenter, toutefois les éléments déclencheur d'un nouveau retrait d'indentation ne se réduisent pas uniquement aux ouvertures/fermetures de bloc et d'instructions du type if...then...else. En outre pour des raisons de choix de présentation une ligne peut s'aligner sur la ligne précédente ou bien plutôt avancer d'une tabulation sur la ligne précédente; dans cette dernière hypothèse nous procédons par **positionnement relatif** et non pas par positionnement absolu.

L'automate "moteur de décision" va donc agir par positionnement **absolu** dans certains cas et par positionnement **relatif** dans d'autres. C'est de la variabilité et du dosage entre ces deux types d'actions que nous obtiendrons une présentation personnalisée.

Ci-dessous l'insertion de la classe indentateur de code dans une éditeur syntaxique :





Exercices chapitre 6

Ex-1 : Soit G la C-grammaire suivante et L(G) le langage engendré par G :

G : V_N = { S, A ,B }
V_T = { **(** , **)** , **o** }
Axiome : A
Règles :
1 : A → **(** A
2 : A → **(** S
3 : S → **o** S
4 : S → **)** B
5 : B → **)** B
6 : B → **)**

1°) construisez l'arbre de dérivation dans G de la chaîne : **(** ^{3} **o**^{2} **)**^{4}

2°) construisez un automate fini reconnaissant le langage L(G) :
2.1) en fournissant son graphe
2.2) en indiquant s'il est déterministe ou non
2.3) donnez la séquence d'analyse de la chaîne **(** ^{3} **o**^{2} **)**^{4}

Ex-2 : On donne la Grammaire G_0 de type 3 suivante :

V_N = { ⟨programme⟩ , ⟨instruction⟩ , ⟨affectation⟩ , ⟨terme1⟩, ⟨ terme2⟩, ⟨membre droit⟩ , ⟨oper⟩ }

V_T = { '**a**' , '**b**' , … ,'**z**' , '**fin**' , '**debut**' , '**;**' , '**Lire**', '**Ecrire**' , '**+**' , '**/**' , '*****' , '**-**' , '**=**' }
*('**a**' , '**b**' , … , '**z**' désigne toutes les lettres de l'alphabet minuscules)*

Axiome : ⟨programme⟩

Règles :
⟨programme⟩ ⟶ **debut** ⟨ instruction⟩
⟨instruction⟩ ⟶ **fin**
⟨instruction⟩ ⟶ **a** ⟨affectation⟩ | **b** ⟨affectation⟩ | ... | **z** ⟨affectation⟩
⟨instruction⟩ ⟶ **Lire** ⟨ terme1⟩ | **Ecrire**⟨ terme1⟩
⟨ terme1⟩ ⟶ **a** ⟨ terme2⟩ | **b** ⟨ terme2⟩ | ... | **z** ⟨terme2⟩
⟨ terme2⟩ ⟶ **;** ⟨instruction⟩
⟨affectation⟩ ⟶ = ⟨membre droit⟩
⟨membre droit⟩ ⟶ **a** ⟨oper⟩ | **b** ⟨oper⟩ | ... | **z** ⟨oper⟩
⟨oper⟩ ⟶ **+** ⟨ terme1⟩ | ***** ⟨ terme1⟩ | **/** ⟨ terme1⟩ | **-** ⟨ terme1⟩ | ⟨ terme2 ⟩

Cette grammaire G_0 engendre un langage noté $L(G_0)$.

Questions :

1°) combien de règles distinctes possède la grammaire G_0 ?

2°) On donne les deux mots suivants *(les blancs ne sont pas significatifs et ne sont figurés que pour rendre le texte lisible)* :

mot_1 = **debut** **Lire** x ; **Lire** y ; z = x * y ; **Lire** a ; a = a - z ; **Ecrire** a ; **fin**

mot_2 = **debut** **Lire** x ; **Lire** y ; **Lire** a ; a = a - x * y ; **Ecrire** a ; **fin**

En construisant leur arbre de dérivation potentiel dans G_0 , dire si ces deux mots appartiennent ou non au langage de $L(G_0)$.

3°) Construire un automate d'état fini reconnaissant $L(G_0)$ indiquez s'il est déterministe ou non.

4°) Donnez une séquence d'analyse (liste des règles de transition appliquées) de chacun des deux mots de la question 2° et confirmez ainsi votre réponse d'appartenance ou non de ces mots au langage $L(G_0)$.

Ex-3 : Analyse d'une expression arithmétique parenthésée et traduction en expression postfixée, par algorithme d'automate avec pile LIFO. On propose les spécifications suivantes :

Une expression est stockée dans une chaîne de caractères.
- Les expressions sont composées de variables, de chiffres et d'opérateurs
- Les variables ne sont composées que d'une lettre (a,b,c,...z)
- Les chiffres sont : (0, 1, 2, ... ,9)
- Les opérateurs sont : (+, - , * , / , ^ , !)
- Les priorités d'opérateurs sont les suivantes :
  + , - : priorité = 1

  / , * : priorité = 2

  ^ : priorité = 3

  ! : priorité = 4

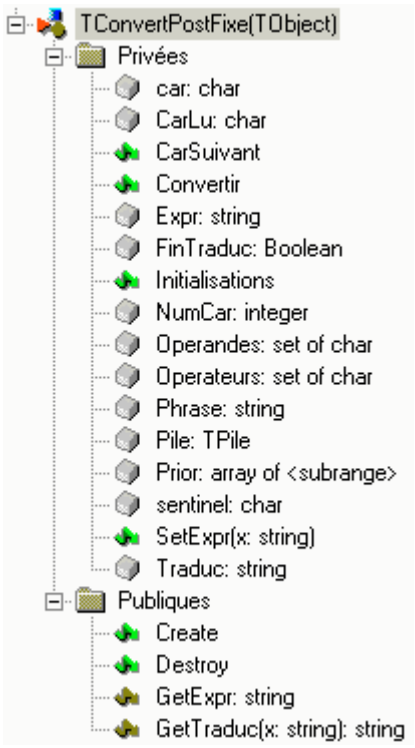
```
Algorithme AutomateParPostFixe
entrée :  phrase //une chaîne contenant l'expression parenthésée
sortie : Traduc //une chaîne contenant l'expression postfixée
local :
 FinAnalyse € booleen,
 CarLu € car,
 sentinelle € car
 Pile //est une pile LIFO contenant des caractères
 EnsdesOpérateurs //ensemble des opérateurs
 EnsdesOpérandes // ensemble des opérandes (variables et chiffres)

debut
 lire(phrase);
 FinAnalyse <--- Faux ;
 CarLu <--- 1er caractère de phrase ;
 tantque non FinAnalyse faire
  si CarLu € EnsdesOpérandes alors
   concaténer CarLu à Traduc ;
   CarLu suivant ;
  sinon
   si CarLu = ( alors
    Empiler CarLu ;
    CarLu suivant ;
   sinon
    si CarLu € EnsdesOpérateurs alors
     si Sommet de Pile = ( ou Pile est vide alors
      Empiler CarLu ;
      CarLu suivant ;
     sinon //le sommet de pile est un opérateur
      si priorité ( CarLu) > priorité(Sommet de Pile) alors
       Empiler CarLu ;
       CarLu suivant ;
      sinon
       concaténer Sommet de Pile à Traduc
       Dépiler ;
```

```
     fsi
    fsi
   sinon
    si CarLu = ) alors
     si Sommet de Pile appartient  EnsdesOpérateurs  alors
      concaténer Sommet de Pile à Traduc
      Dépiler ;
     sinon //sommet de Pile = (
      Dépiler ;
      CarLu suivant ;
     fsi
    sinon
     si CarLu = sentinelle alors
      si Pile n'est pas vide alors
       concaténer Sommet de Pile à Traduc
       Dépiler ;
      sinon
       FinAnalyse <--- Vrai ;
      fsi
     sinon
      Erreur
     fsi
    fsi
   fsi
  fsi
 fsi
ftant
fin // AutomateParPostFixe
```

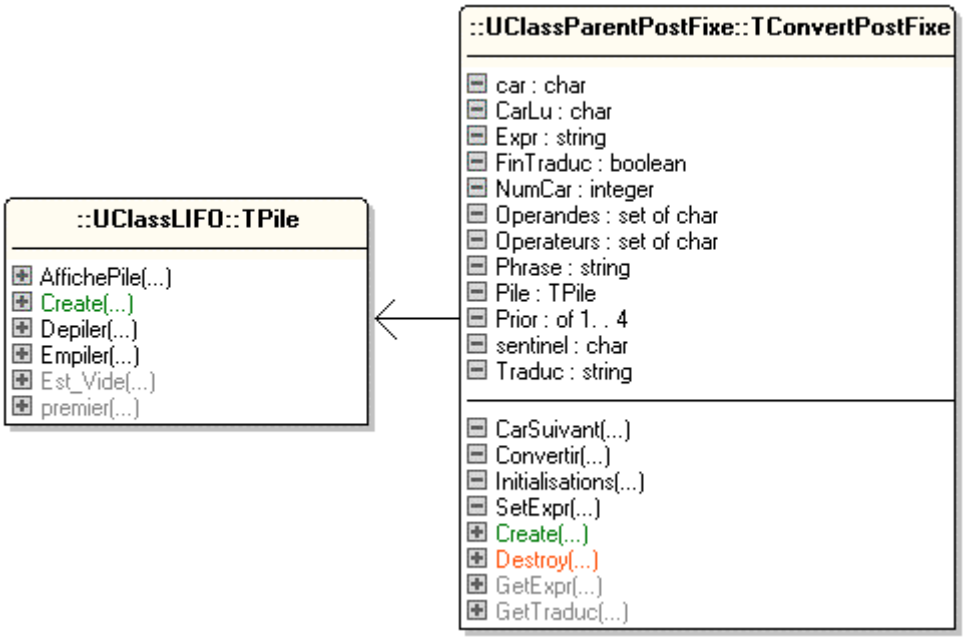
Questions :

1°) Effectuez une trace formelle à figurer dans un tableau de cet algorithme sur l'expression suivante : **(a+b)*c-5**
.

2°) Ci-dessous voici une signature d'une implémentation possible de cet algorithme avec deux classes en Delphi :



On suppose que la classe TPile est déclarée et entièrement implémentée dans la Unit UClassLIFO :



Il vous est demandé de donner le contenu détaillé de la Unit UclassParentPostFixe contenant la classe TconvertPostFixe qui implante l'algorithme précédent.

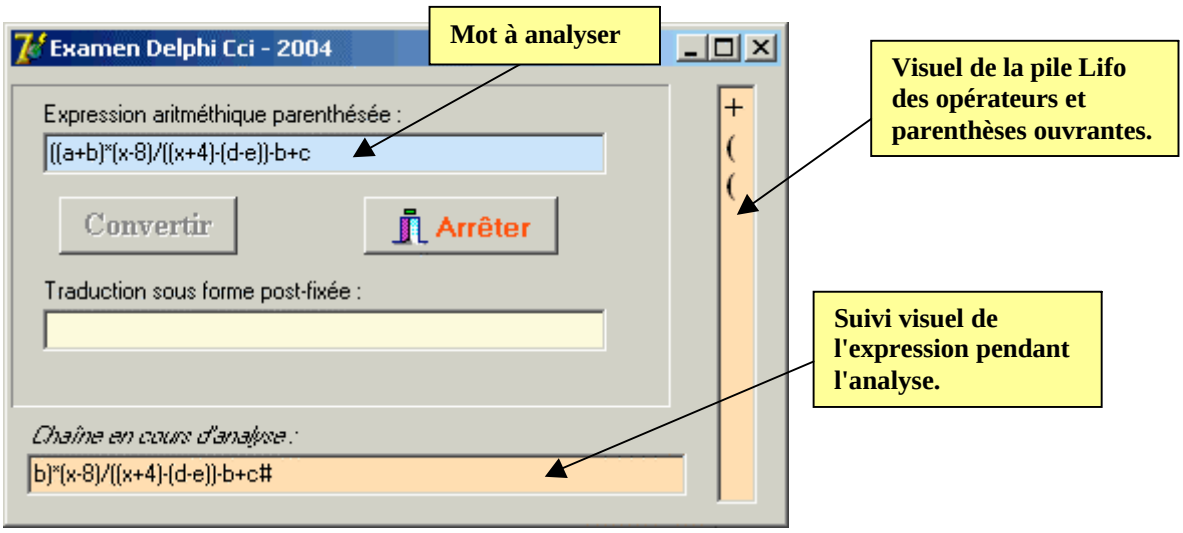
Quelques remarques utiles pour votre implémentation:

Les données:
Expr contient l'expression parenthésée telle qu'elle est rentrée,
Phrase contient l'expression à analyser avec la sentinelle,
Traduc contient l'expression postfixée.

Les méthodes :
GetExpr renvoie l'expression telle qu'elle a été rentrée,
GetTraduc renvoie l'expression une fois traduite en postfixé,
SetExpr prépare les données pour le lancement de la conversion,
Initialisations initialise toutes les données : rempli la table de priorités, les opérateurs, les opérandes…
Convertir : l'algorithme de conversion proprement dit.

3°) On demande de construire une interface d'animation du suivi de l'analyse d'une expression arithmétique.

Cette interface est développée pour tester l'analyseur construit aux questions précédentes. Elle permet d'entrer une expression arithmétique juste, puis de lancer la conversion, alors s'affichent à chaque caractère analysé le contenu de la chaîne en cours d'analyse et le contenu de la pile Lifo des opérateurs et parenthèses ouvrantes.



Travail a effectuer :

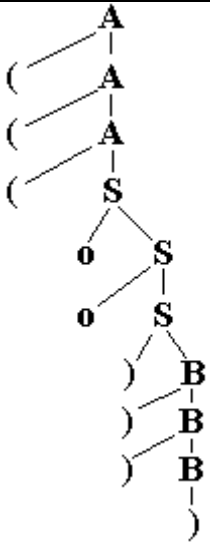
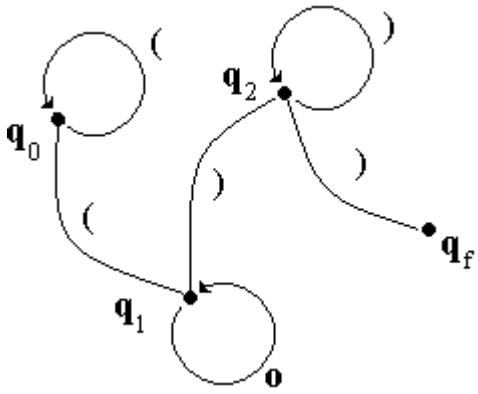
Utiliser dans la classe **TconvertPostFixe** l'agrégation forte d'un objet de classe **TTimer** pour obtenir un programme interactif au sens suivant :

- On peut fermer la fiche en cours d'analyse (à tout moment)
- On peut arrêter à tout moment l'animation de la traduction et la phase de traduction se poursuit alors en mode exécutable sans temporisation.

Rendre un objet de classe **TconvertPostFixe** sensible à un événement que vous dénommerez OnEndTraduction qui se déclenche dès que la traduction de l'expression est terminée, puis dans l'IHM de test interceptez cet événement afin qu'il affiche un message dans une fenêtre de showmessage.

Réponses

Ex-1 :

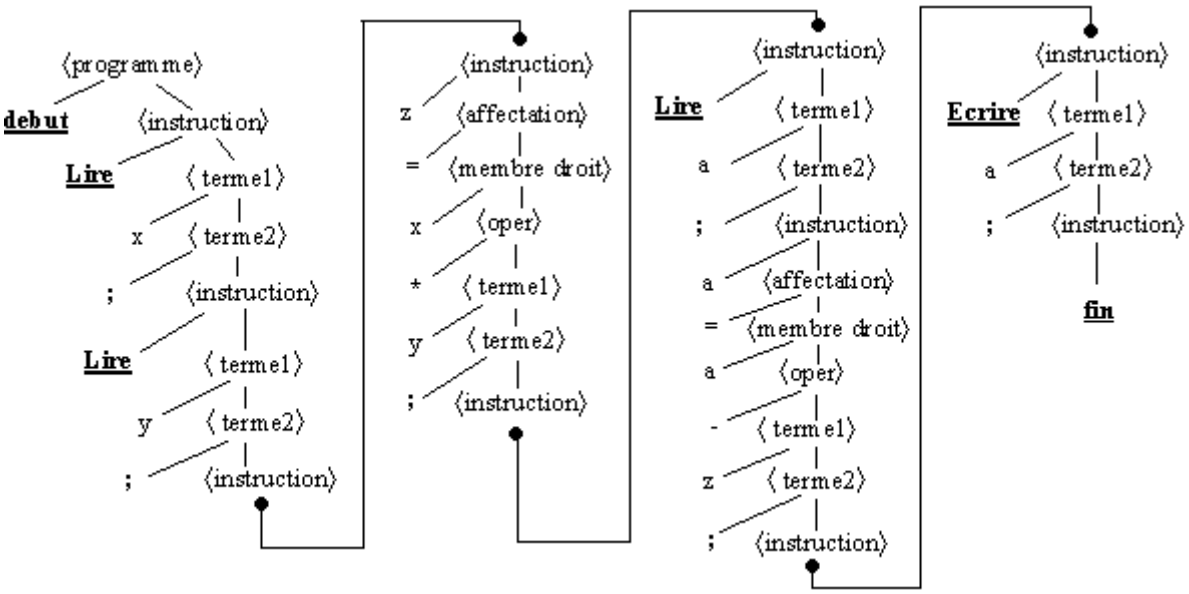
Arbre de dérivation de $({}^{3}\ \mathbf{o}^{2}\ )^{4}$	Automate d'état fini reconnaissant L(G)
	 L'automate est non déterministe. Séquence d'analyse de $({}^{3}\ \mathbf{o}^{2}\ )^{4}$: $(\mathbf{q}_0, '(') \rightarrow \mathbf{q}_0$; $(\mathbf{q}_0, '(') \rightarrow \mathbf{q}_0$; $(\mathbf{q}_0, '(') \rightarrow \mathbf{q}_1$; $(\mathbf{q}_1, 'o') \rightarrow \mathbf{q}_1$; $(\mathbf{q}_1, 'o') \rightarrow \mathbf{q}_1$; $(\mathbf{q}_1, ')') \rightarrow \mathbf{q}_2$; $(\mathbf{q}_2, ')') \rightarrow \mathbf{q}_2$; $(\mathbf{q}_2, ')') \rightarrow \mathbf{q}_2$; $(\mathbf{q}_2, ')') \rightarrow \mathbf{q}_f$

Ex-2 :

1°) Règles :

⟨programme⟩ ⟶ **debut** ⟨ instruction⟩	= 1 règle
⟨instruction⟩ ⟶ **fin**	= 1 règle
⟨instruction⟩ ⟶ **a** ⟨affectation⟩ | **b** ⟨affectation⟩ | ... | **z** ⟨affectation⟩	= 26 règles
⟨instruction⟩ ⟶ **Lire** ⟨ terme1⟩ | **Ecrire**⟨ terme1⟩	= 2 règles
⟨ terme1⟩ ⟶ **a** ⟨ terme2⟩ | **b** ⟨ terme2⟩ | ... | **z** ⟨terme2⟩	= 26 règles
⟨ terme2⟩ ⟶ **;** ⟨instruction⟩	= 1 règle
⟨affectation⟩ ⟶ = ⟨membre droit⟩	= 1 règle
⟨membre droit⟩ ⟶ **a** ⟨oper⟩ | **b** ⟨oper⟩ | ... | **z** ⟨oper⟩	= 26 règles
⟨oper⟩ ⟶ + ⟨ terme1⟩ | * ⟨ terme1⟩ | / ⟨ terme1⟩ | - ⟨ terme1⟩ | ⟨ terme2 ⟩	= 5 règles
	= 26 règles
Total	= 115 règles

2°) Arbre de dérivation de mot_1 = **debut Lire** x ; **Lire** y ; z = x * y ; **Lire** a ; a = a - z ; **Ecrire** a ; **fin**



celui de mot_2 ne peut être construit en entier car l'instruction a = a - x * y ; n'est pas dérivable dans la grammaire.
Donc mot_1 appartient à $L(G_0)$, mot n'appartient pas à $L(G_0)$.

3°)

AEF donné par ses règles

(q_0 , **debut**) → q_1	(q_6 , *Operat*) → q_3	*Lettres = { a, b, … , z }*
(q_1 , **fin**) → q_f	(q_6 , ;) → q_1	*Operat = { +,- , /,* }*
(q_1 , *Lettres*) → q_2	(q_1 , **Lire**) → q_3	
(q_2 , =) → q_5	(q_1 , **Ecrire**) → q_3	
(q_5 , *Lettres*) → q_6	(q_3 , *Lettres*) → q_4	
(q_4 , ;) → q_1		

Reconnaissance de **mot_1**

(q_0 , **debut**) → q_1	(q_3 , y) → q_4	(q_6 , *) → q_3	(q_4 , ;) → q_1	(q_3 , z) → q_4
(q_1 , **Lire**) → q_3	(q_4 , ;) → q_1	(q_3 , y) → q_4	(q_1 , a) → q_2	(q_4 , ;) → q_1
(q_3 , x) → q_4	(q_1 , z) → q_2	(q_4 , ;) → q_1	(q_2 , =) → q_5	(q_1 , **Ecrire**) → q_3
(q_4 , ;) → q_1	(q_2 , =) → q_5	(q_1 , **Lire**) → q_3	(q_5 , a) → q_6	(q_3 , y) → q_4
(q_1 , **Lire**) → q_3	(q_5 , x) → q_6	(q_3 , a) → q_4	(q_6 , -) → q_3	(q_4 , ;) → q_1
				(q_1 , **fin**) → q_f

Tentative de reconnaissance de **mot_2**			
(q_0 , **debut**) → q_1 (q_1 , **Lire**) → q_3 (q_3 , x) → q_4 (q_4 , **;**) → q_1 (q_1 , **Lire**) → q_3	(q_3 , y) → q_4 (q_4 , **;**) → q_1 (q_1 , **Lire**) → q_3 (q_3 , a) → q_4 (q_4 , **;**) → q_1	(q_1 , a) → q_2 (q_2 , =) → q_5 (q_5 , a) → q_6 (q_6 , -) → q_3 (q_3 , x) → q_4	(q_4 , *****) → **??** aucune règle de la forme (q_4 , *****) → … Donc mot_2 n'est pas reconnu.

Remarquons que mot_2 est partiellement conforme à la syntaxe jusqu'à :
debut **Lire** x ; **Lire** y ; **Lire** a ; a = a - x ← le reste n'est pas correct

Ex-3 : Solution pratique de "expression arithmétique parenthésée"

Questions 2°)

```
unit UClassLIFO;
{une implantation en Delphi du TAD Pile LIFO
à partir de la classe tlist     }

interface

 uses classes;

 type
  TPile = class(TList)
 public
  constructor Create;
  function  Est_Vide : boolean;
  procedure Empiler (elt: char);
  procedure Depiler (var elt: char);
  function  premier : char;
  procedure AffichePile;
end;


implementation

 constructor TPile.Create;
 begin
  inherited Create;
 end;

 function TPile.Est_Vide: boolean;
 begin
  if count = 0 then
   result := true
  else
   result := false
 end;

 procedure TPile.Empiler (elt: char);
 begin
  self.Capacity := Count;
  Add(TObject(elt))
 end;
```

```
procedure TPile.Depiler (var elt: char);
begin
 if not Est_Vide then
 begin
  elt:=char(Last);
  self.Delete(count-1);
  self.Pack;
  self.Capacity := Count;
 end
else
  writeln(' désolé pile vide !');
end;

function TPile.premier : char;
begin
 if not est_vide then
  result:=char(last)
 else
  result:='@';
end;

procedure TPile.AffichePile;
 var i:integer;
 begin
  if not Est_Vide then
  begin
   writeln('Pile contenant : ',count,' éléments.');
   for i:=0 to count-1 do
    write(char(Items[i]));
   writeln
  end
  else
   writeln('pile vide.');
 end;

end.

unit UClassParentPostFixe;
{implantation en Delphi d'un convertisseur d'expression parenthésée en postfixé
avec pile lifo       }

interface

 uses UClassLIFO;

 type
{ Grammaire :
    opérandes possibles: a,b,...,z,0,1,...,9
    opérateurs possibles: +,-,/,*,^,!
      +,- : prior = 1

      /,* : prior = 2

       ^ : prior = 3

       ! : prior = 4
}
 TConvertPostFixe = class
 private
  Pile:TPile;
```

Ecriture d'informations sur la console.

```
  Phrase,Expr,Traduc:string;
  CarLu,car,sentinel:char;
  NumCar:integer;
  Operandes,Operateurs:set of char;
  Prior:array[char] of 1..4;
  FinTraduc:Boolean;
  procedure Initialisations;
  procedure CarSuivant;
  procedure SetExpr(x:string);
  procedure Convertir;
 public
  constructor Create;
  destructor Destroy;
  function GetExpr:string;
  function GetTraduc(x:string):string;
end;

implementation
uses Dialogs;

 // Private --->
 procedure TConvertPostFixe.Initialisations;
 begin
  Prior['+']:=1;
  Prior['-']:=1;
  Prior['/']:=2;
  Prior['*']:=2;
  Prior['^']:=3;
  Prior['!']:=4;
  Operandes:=['a'..'z']+['0'..'9'];
  Operateurs:=['+','-','/','*','^','!'];
  NumCar:=0;
  Traduc:='';
  FinTraduc:=false;
 end;

 procedure TConvertPostFixe.CarSuivant;
 begin
  NumCar:=NumCar+1;
  CarLu:=Phrase[NumCar]
 end;

 procedure TConvertPostFixe.SetExpr(x:string);
 begin
  Traduc:='';
  Expr:=x;
  Phrase:=Expr+sentinel;
 end;

 procedure TConvertPostFixe.Convertir;
 begin
  CarSuivant;
  while not FinTraduc do
  begin
   if CarLu in Operandes then
   begin
    Traduc:=Traduc+CarLu;
    CarSuivant;
   end
   else
```

```
  if carlu='(' then
  begin
   Pile.Empiler(CarLu);
   CarSuivant;
  end
  else
   if carlu in operateurs then
   begin
    if (Pile.premier='(')or(Pile.Est_Vide) then
    begin
     Pile.Empiler(CarLu);
     CarSuivant;
    end
    else {sommet de pile est un opérateur}
     if Prior[CarLu] > Prior[Pile.premier] then
     begin
      Pile.Empiler(CarLu);
      CarSuivant;
     end
     else
     begin
      Traduc:=Traduc+Pile.premier;
      Pile.Depiler(car);
     end
    end
    else
     if carlu=')' then
     begin
      if Pile.premier in Operateurs then
      begin
       Traduc:=Traduc+Pile.premier;
       Pile.Depiler(car);
      end
      else {sommet de pile = '('}
      begin
       Pile.Depiler(car);
       CarSuivant;
      end
     end
     else
      if carlu = sentinel then
      begin
       if not Pile.Est_Vide then
       begin
        Traduc:=Traduc+Pile.premier;
        Pile.Depiler(car);
       end
       else
        fintraduc:=true;
      end
 end;
end;

// Public --->
constructor TConvertPostFixe.Create;
begin
 sentinel:='#';
 Pile:=TPile.Create;
 Initialisations;
end;
```

```
destructor TConvertPostFixe.Destroy;
begin
 Pile.Free;
 Pile:=nil;
 inherited;
end;

function TConvertPostFixe.GetExpr:string;
begin
 result:= Expr
end;

function TConvertPostFixe.GetTraduc(x:string):string;
begin
 SetExpr(x);
 Convertir;
 if length(traduc)=0 then
  showmessage('expression vide');
 result:=Traduc
end;

end.
```

Version IHM et animation

```
unit UClassLIFO;
{une implantation en Delphi du TAD Pile LIFO
à partir de la classe tlist    }

interface

 uses classes;

 type
  TPile = class(TList)
 public
  constructor Create;
  function  Est_Vide : boolean;
  procedure Empiler (elt: char);
  procedure Depiler (var elt: char);
  function  premier : char;
 // procedure AffichePile;
end;


implementation

 constructor TPile.Create;
 begin
  inherited Create;
 end;

 function TPile.Est_Vide: boolean;
 begin
  if count = 0 then
   result := true
  else
```

```
  result := false
end;

procedure TPile.Empiler (elt: char);
begin
 self.Capacity := Count;
 Add(TObject(elt))
end;

procedure TPile.Depiler (var elt: char);
begin
 if not Est_Vide then
 begin
  elt:=char(Last);
  self.Delete(count-1);
  self.Pack;
  self.Capacity := Count;
 end
// else
//  writeln(' désolé pile vide !');
end;

function TPile.premier : char;
begin
 if not est_vide then
  result:=char(last)
 else
  result:='@';
end;
{
procedure TPile.AffichePile;
var i:integer;
begin
 if not Est_Vide then
 begin
  writeln('Pile contenant : ',count,' éléments.');
  for i:=0 to count-1 do
   write(char(Items[i]));
  writeln
 end
 else
  writeln('pile vide.');
end;
 }
end.
```

On enlève dans la classe précédente tout relation avec une écriture sur la console.

```
unit UClassParentPostFixe;
{implantation en Delphi d'un convertisseur d'expression parenthésée correcte en postfixé avec pile lifo      }

interface

 uses UClassLIFO,Classes,StdCtrls,SysUtils,Forms,ExtCtrls;

 type
 {
    opérandes possibles: a,b,...,z,0,1,...,9
    opérateurs possibles: +,-,/,*,^,!
      +,- : prior = 1
      /,* : prior = 2
```

```
        ^ : prior = 3
        ! : prior = 4
}
TConvertPostFixe = class(TComponent)
 private
  FonEndTraduction:TNotifyEvent;
  dureefinie:boolean;
  Farret:boolean;
  FTimer:TTimer; //agrégation forte
  Flifo:TListBox; // référence vers un TListBox : agrégation faible
  PhraseEncours:Tedit; // référence vers un TEdit : agrégation faible
  Pile:TPile;
  Phrase,Expr,Traduc:string;
  CarLu,car,sentinel:char;
  NumCar:integer;
  Operandes,Operateurs:set of char;
  Prior:array[char] of 1..4;
  FinTraduc:Boolean;
  procedure Initialisations;
  procedure CarSuivant;
  procedure SetExpr(x:string);
  procedure Convertir;
  procedure Temporiser;
  function Gettempo:integer;
  procedure Settempo(x:integer);
  procedure OnTimerFTimer(Sender:TObject);
  procedure Setarret(x:boolean);
  function Getarret:boolean;
 public
  constructor Create(lifo:TListBox;listc:TEdit);reintroduce;overload;
  constructor Create(lifo:TListBox;listc:TEdit;temps:integer);reintroduce;overload;
  destructor Destroy;override;
  function GetExpr:string;
  function GetTraduc(x:string):string;
 published
  property tempo:integer read Gettempo write Settempo;
  property arret:boolean read Getarret write Setarret;
  property OnEndTraduction:TNotifyEvent read FonEndTraduction write FonEndTraduction;
end;

implementation
uses Dialogs;

 // Private --->
 procedure TConvertPostFixe.Initialisations;
 begin
  Prior['+']:=1;
  Prior['-']:=1;
  Prior['/']:=2;
  Prior['*']:=2;
  Prior['^']:=3;
  Prior['!']:=4;
  Operandes:=['a'..'z']+['0'..'9'];
  Operateurs:=['+','-','/','*','^','!'];
  NumCar:=0;
  Traduc:='';
  FinTraduc:=false;
  Farret:=false;
 end;
```

```
procedure TConvertPostFixe.CarSuivant;
begin
 NumCar:=NumCar+1;
 CarLu:=Phrase[NumCar];
 PhraseEncours.Text:=copy(Phrase,NumCar,length(Phrase));
 if not Farret then
  Temporiser; // temporisation pour animation
end;

procedure TConvertPostFixe.SetExpr(x:string);
begin
 Traduc:='';
 Expr:=x;
 Phrase:=Expr+sentinel;
end;

procedure TConvertPostFixe.Convertir;
begin
 CarSuivant;
 while not FinTraduc do
 begin
  if CarLu in Operandes then
  begin
   Traduc:=Traduc+CarLu;
   CarSuivant;
  end
  else
   if carlu='(' then
   begin
    Pile.Empiler(CarLu);
    CarSuivant;
   end
   else
    if carlu in operateurs then
    begin
     if (Pile.premier='(')or(Pile.Est_Vide) then
     begin
      Pile.Empiler(CarLu);
      CarSuivant;
     end
     else {sommet de pile est un opérateur}
      if Prior[CarLu] > Prior[Pile.premier] then
      begin
       Pile.Empiler(CarLu);
       CarSuivant;
      end
      else
      begin
       Traduc:=Traduc+Pile.premier;
       Pile.Depiler(car);
      end
    end
    else
     if carlu=')' then
     begin
      if Pile.premier in Operateurs then
      begin
       Traduc:=Traduc+Pile.premier;
       Pile.Depiler(car);
      end
```

```
     else {sommet de pile = '('}
     begin
      Pile.Depiler(car);
      CarSuivant;
     end
    end
    else
     if carlu = sentinel then
     begin
      if not Pile.Est_Vide then
      begin
       Traduc:=Traduc+Pile.premier;
       Pile.Depiler(car);
      end
      else
      begin
       FinTraduc:=true;
       if Assigned(FOnEndTraduction)then
        FOnEndTraduction(self)
      end
     end
 end;
end;

// Public --->
constructor TConvertPostFixe.Create(lifo:TListBox;listc:TEdit);
begin
 sentinel:='#';
 Flifo:=lifo;
 dureefinie:=true;
 PhraseEncours:= listc;
 Pile:=TPile.Create(lifo);
 FTimer:=TTimer.Create(self);
 FTimer.Interval:=1000;
 FTimer.Enabled:=false;
 FTimer.OnTimer:=OnTimerFTimer;
 Initialisations;
end;

constructor TConvertPostFixe.Create(lifo:TListBox;listc:TEdit;temps:integer);
begin
 sentinel:='#';
 Flifo:=lifo;
 dureefinie:=true;
 PhraseEncours:= listc;
 Pile:=TPile.Create(lifo);
 FTimer:=TTimer.Create(self);
 FTimer.Interval:=temps;
 FTimer.Enabled:=false;
 FTimer.OnTimer:=OnTimerFTimer;
 Initialisations;
end;

destructor TConvertPostFixe.Destroy;
begin
 Pile.Free;
 Pile:=nil;
 inherited;
end;
```

OnEndTraduction est donc déclenché dès que la traduction de l'expression est terminée.

```
function TConvertPostFixe.GetExpr:string;
begin
 result:= Expr
end;

function TConvertPostFixe.GetTraduc(x:string):string;
begin
 Initialisations;
 SetExpr(x);
 Convertir;
 if length(traduc)=0 then
  showmessage('expression vide');
 result:=Traduc
end;

procedure TConvertPostFixe.Temporiser;
begin
 dureefinie:=false;
 FTimer.Enabled:=true;
 repeat
  Application.ProcessMessages
 Until  (ftimer.enabled=false)or(application.terminated);
 Flifo.Repaint;
 PhraseEncours.Repaint;
end;

function TConvertPostFixe.Gettempo: integer;
begin
 result:=FTimer.Interval
end;

procedure TConvertPostFixe.Settempo(x: integer);
begin
 if x>0 then
  FTimer.Interval:=x
 else
  FTimer.Interval:=100
end;

procedure TConvertPostFixe.OnTimerFTimer(Sender: TObject);
begin
 if dureefinie=true then
  FTimer.Enabled:=false
 else
  dureefinie:=true
end;

function TConvertPostFixe.Getarret: boolean;
begin
 result:=Farret;
end;

procedure TConvertPostFixe.Setarret(x: boolean);
begin
 Farret:=x;
 FTimer.Enabled:=not x;
end;

end.
```

Boucle infinie arrêtée par le Timer.