Au menu de l'UE Compilation

III - Programmation d'un automate fini déterministe

Type d'automate / type de programmation :

- **Analyseurs lexical et syntaxique -> automates finis et déterministes**

- **Analyseur lexical :**
  - **Automate simple :** *i.e. presque un* seul chemin d'un état intermédiaire à l'état final
  - **Mise en œuvre : Programmation directe**

- **Analyseur syntaxique :**
  - **Automate complexe** : *i.e.* nombre d'états, de transitions et actions plus importants
  - **Mise en œuvre : Programmation par interpréteur de tables**

A - Programmation par automates finis

III - Programmation d'un automate fini déterministe

III.1 – Programmation directe

III.2 – Programmation par interpréteur de tables

III.3 – Traitement des erreurs dans l'analyse syntaxique

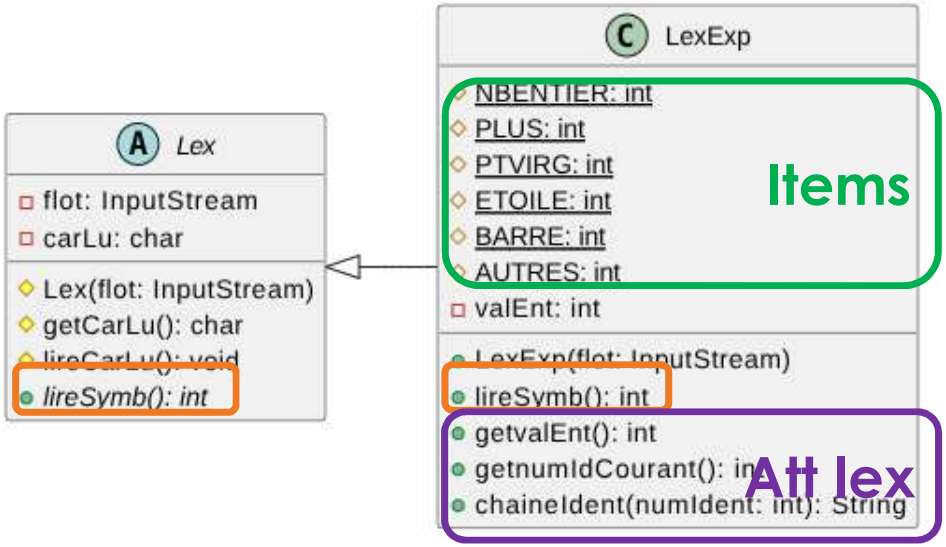
III - Programmation d'un automate fini déterministe

III.1 – Programmation directe

- **Programmation directe** si automate = union d'automates « quasi-linéaires »
  - Ex : automate d'analyse lexicale

Principe :

- Une classe abstraite *Lex*, avec :
  - Méthode abstraite lireSymb()
- Une classe, héritant de Lex, **par langage** (par V_T à traiter)
  - Implémente lireSymb pour reconnaitre les items
  - Définit les attributs lexicaux nécessaires
  - Exemple LexExp : *cf.* Moodle



III - Programmation d'un automate fini déterministe

III.1 – Programmation directe

- Chaque implémentation de la classe Lex sera **spécifique au V_T à traiter et aux items à reconnaitre**
- **Exemple : pour une suite d'expressions LexExp (*cf.* Moodle)**
  - V_T = { *0..9*, +, *, ; , /, *espace* }
  - Traitement des entiers délégué à une méthode

```
/** Lecture d'un item NBENTIER
  * et mise à jour de l'attribut lexical valEnt.
  * @return code NBENTIER */
private int lireEnt() {
  String s = "";
  do {
    s = s + getCarLu();
    lireCarLu();
  } while ((getCarLu() >= '0') && (getCarLu() <= '9'));
  valEnt = Integer.parseInt(s);
  return NBENTIER;
}
```

Exercice pour le langage Monnaie :

Donner la spécification de **private int lireIdent()**

```
/** Lecture du prochain item lexical.
 * Maintien du caractère d'avance connu.
 * @return code de l'item lexical reconnu
 */
public int lireSymb() {
 // On ignore les espaces et assimilés.
  while (getCarLu() == ' ') {
     lireCarLu();
  }
  // On détecte le début de l'item lexical IDENT
  if ((getCarLu() >= '0') && (getCarLu() <= '9')) {
    return lireEnt();
  }
  // On détecte un autre item lexical
  switch (getCarLu()) {
    case '+': lireCarLu(); return PLUS;
    case ';': lireCarLu(); return PTVIRG;
    case '*': lireCarLu(); return ETOILE;
    case '/': return BARRE;
    default : System.out.println("Caractère incorrect");
              lireCarLu();
              return AUTRES;
  }
}
```

Exercice à préparer pour le TP

- Nous avons vu la spécification de la méthode ***private int lireIdent()*** pour le langage Monnaie où V_T = {a..z, A..Z, *0..9*, . , +, -, ; , /, *espace*, ↲ }

Exercice de préparation pour le 1er TP

Rédigez sur feuille la fonction lireIdent pour le langage Monnaie.
On utilisera une fonction similaire pour lire les identifiants et mots-clefs réservés du langage du TP.

A - Programmation par automates finis

III - Programmation d'un automate fini déterministe

III.1 – Programmation directe

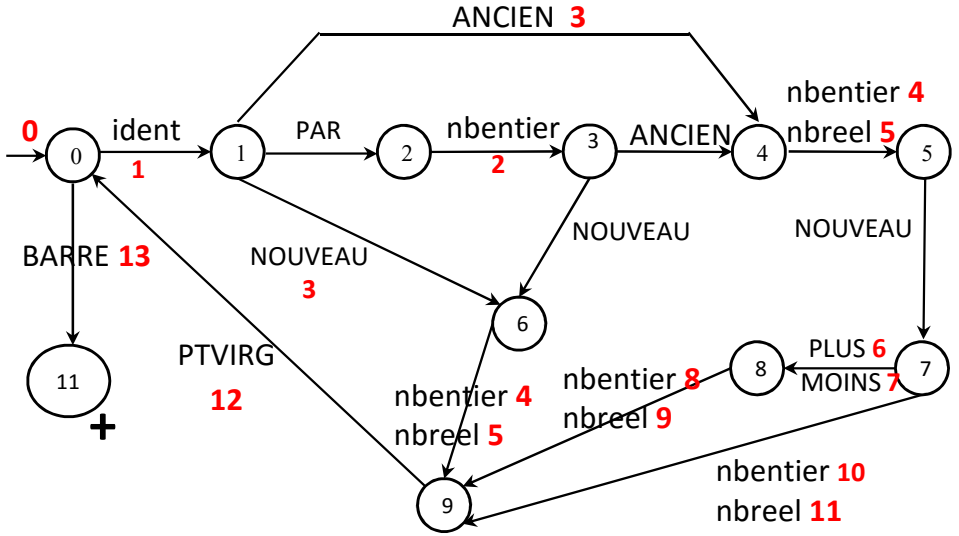
III.2 – Programmation par interpréteur de tables

III.3 – Traitement des erreurs dans l'analyse syntaxique

III - Programmation d'un automate fini déterministe

III.2 – Programmation par interpréteur de tables

- **Automate complexe -> Analyseur syntaxique :**
  - Nb transitions-actions plus importants / état
  - Programmation directe trop compliquée
- **Traitement des automates complexes :**
  - Programmation par interpréteur de tables

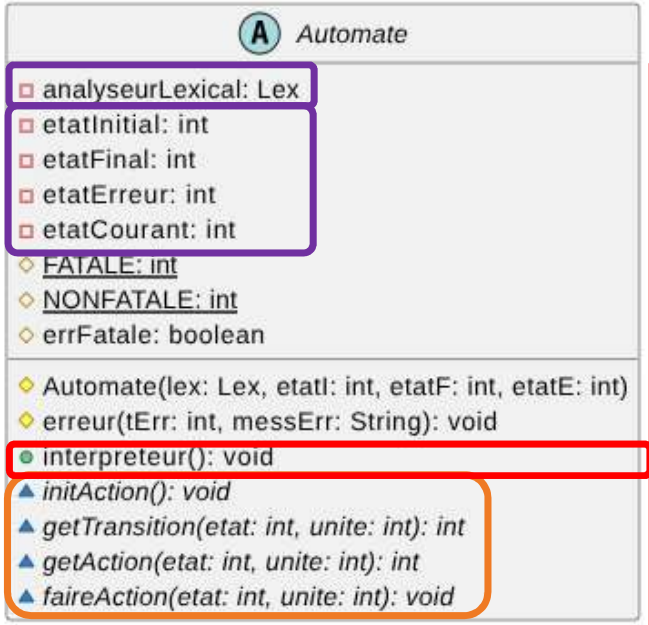


III - Programmation d'un automate fini déterministe

III.2 – Programmation par interpréteur de tables

Principe :

- Classe ***abstraite Automate***
- **Attributs de base :**
  - Un analyseur lexical
  - Les états de l'automate
- Méthode **interpreteur**
  - basée sur les méthodes abstraites initAction, getTransition, faireAction
  - Transitions et actions représentées à l'aide de **tableaux dans les classes filles**



Méthode interpréteur

```
public final void interpreteur() {
  etatCourant = etatInitial; // init état courant
  initAction(); // init nécessaires aux actions
  int token; // unité lexicale courante
  while (etatCourant != etatFinal && !errFatale) {
    token = analyseurLexical.lireSymb();
    int etatDepart = etatCourant;
    faireAction(etatCourant, token);
    etatCourant = getTransition(etatCourant, token);
  }
}
```

III - Programmation d'un automate fini déterministe

III.2 – Programmation par interpréteur de tables

- **C**lasse abstraite *Automate* définit la méthode interpréteur de tables

- **Classe abstraite** *Autoxxx* **définissant l'automate d'analyse syntaxique** (reconnaissance)
  Propre au langage considéré, par ex *AutoExp*
  - hérite de la classe *Automate*
  - définit la table de correspondance (état-courant, code-item-lu) -> état-suivant
  - définit la méthode *getTransition*

- **Classe** *Actxxx* **définissant les actions associées** à l'automate syntaxique
  Propre à l'application considérée, par ex *ActExp*
  - hérite de la classe *Autoxxx*
  - définit la table de correspondance (état-courant, code-item-lu) -> num-action-a-effectuer
  - définit les méthodes *initAction*, *getAction*, *faireAction*

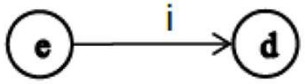
III - Programmation d'un automate fini déterministe

III.2 – Programmation par interpréteur de tables

- Pour l'analyse syntaxique, classe Autoxxx (AutoExp par ex)

- Attribut :
  - table de correspondance (état-courant, code-item-lu) -> état-suivant : **int[][]TRANSIT**

- Pour la transition:



  - **i** ∈ V_T ∪ {autres}, **e** ∈ état : **TRANSIT[e, i] = d**
  - **d est l'état cible**

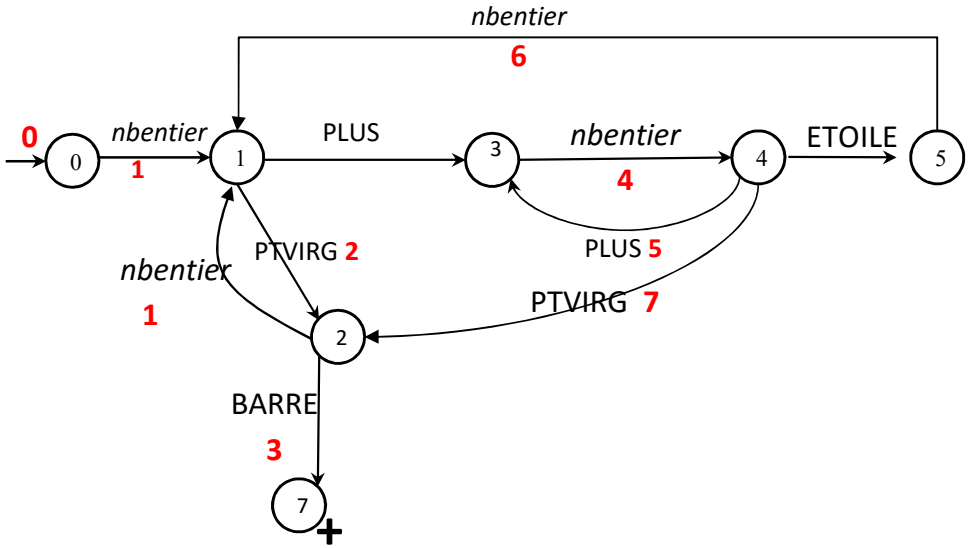
- Ex : *cf.* Moodle autoExpr.zip

- Pour les actions associées, classe Actxxx (ActExp par ex)

- Attribut :
  - table de correspondance (état-courant, code-item-lu) -> num-action-a-effectuer : **int[][]ACTION**

- Pour l'action numéro a :
  - **i** ∈ V_T ∪ {autres}, e ∈ état : **ACTION[e, i] = a**
  - **a est une action**

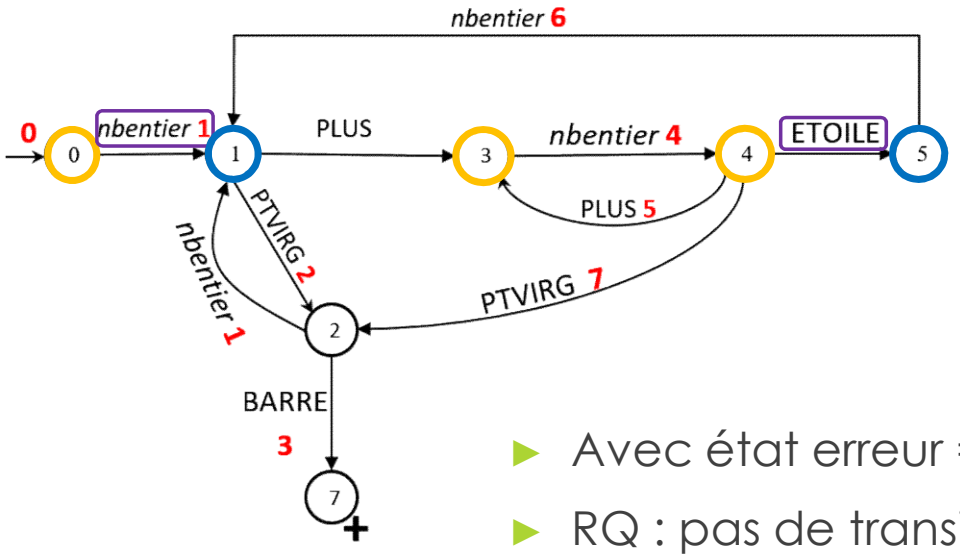
Exemple : automate avec actions des expressions arithmétiques



III - Programmation d'un automate fini déterministe

III.2 – Programmation par interpréteur de tables

- Ex : analyse syntaxique - AutoExp
  - **i** ∈ V_T ∪ {autres}, **e** ∈ état : **TRANSIT[e, i] = d**
  - **d est l'état cible**



TRANSIT

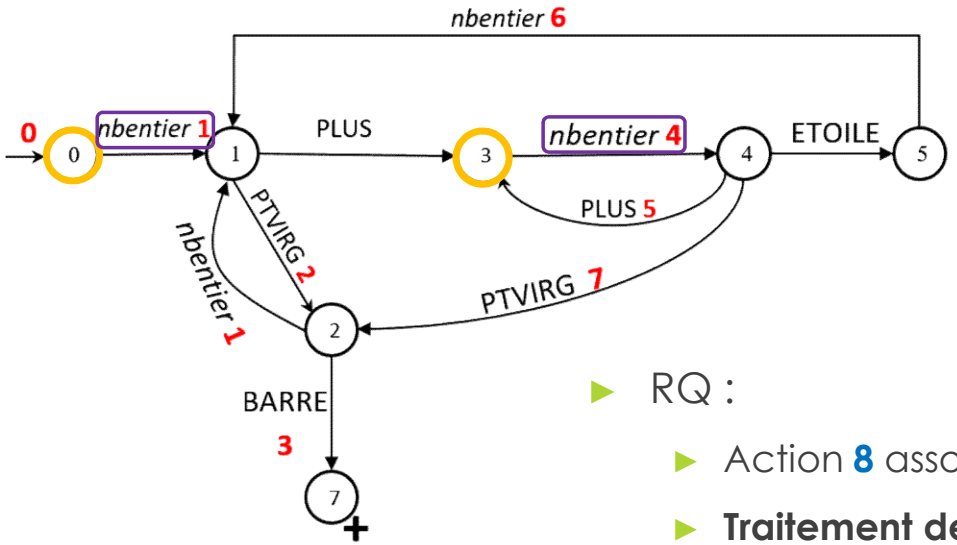
Items / états	NBENTIER 0	PLUS 1	PTVIRG 2	ETOILE 3	BARRE 4	AUTRES 5
0	1	6	6	6	6	6
1	6	3	2	6	6	6
2	1	6	6	6	7	6
3	4	6	6	6	6	6
4	6	3	2	5	6	6
5	1	6	6	6	6	6
etatErreur=6	6	6	6	6	6	6

- Avec état erreur = **6**
- RQ : pas de transition à partir de l'état final 7 => pas de ligne associée

III - Programmation d'un automate fini déterministe

III.2 – Programmation par interpréteur de tables

- Ex : actions - ActExp :
  - **i** ∈ V_T ∪ {autres}, e ∈ état : **ACTION[e, i] = a**
  - **a est une action**



ACTION

Items (codes) / états	NBENTIER (0)	PLUS (1)	PTVIRG (2)	ETOILE (3)	BARRE (4)	AUTRES (5)
0	1	8	8	8	8	8
1	8	-1	2	8	8	8
2	1	8	8	8	3	8
3	4	8	8	8	8	8
4	8	?		-1	8	8
5	6	8	8	8	8	8
etatErreur=6	-1	-1	-1	-1	-1	-1

- RQ :
  - Action **8** associée aux transitions vers l'état erreur 6
  - **Traitement des erreurs de syntaxe**.

III - Programmation d'un automate fini déterministe

III.2 – Programmation par interpréteur de tables

- Implémenter les méthodes abstraites dans les classes héritant d'Automate
  - AutoExp :

    getTransition(etatCourant, unite)

  - ActExp :

    faireAction(etatCourant, unite)
    + initAction()
    + getAction(etatCourant, unite)

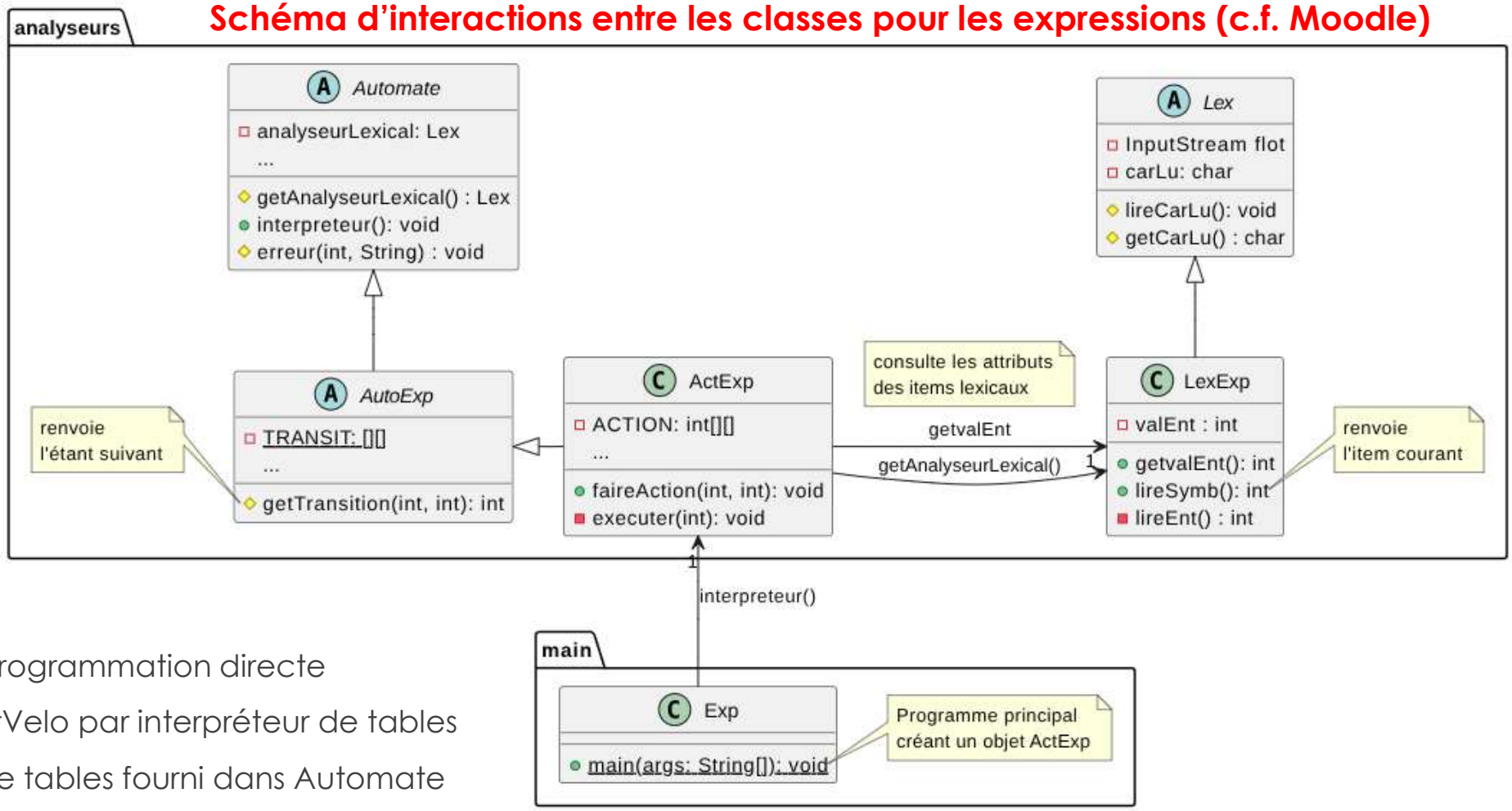
- ActExp :

```java
@Override
public void faireAction(int etat, int itemLex) {
   executer(ACTION[etat][itemLex]);
}
/** Exécution d'une action dans l'état courant
 * @param numact numéro de l'action à exécuter
 */
private void executer(int numAction) {
  switch (numAction) {
    case -1: break; // action vide
    case 0: max = -1; break; // action faite à l'initialisation
    case 1: valExp = analyseurLexical.getvalEnt(); break; // juste un entier
    case 2: finExp(valExp); break; // fin expression sur PTVIRG
    case 3: System.out.println("Le max est " + max); break; // fin donnée
    case 4: // opérande droit de PLUS est un entier peut-être multiplié
            operande = analyseurLexical.getvalEnt(); break;
    case 5: // opérande droit de PLUS est réduit à un entier
            valExp = valExp + operande; break;
    case 6: // opérande droit de PLUS est un produit
            operande = operande * analyseurLexical.getvalEnt();
            valExp = valExp + operande ; break;
    case 7: finExp(valExp + operande); break; // fin addition sur PTVIRG
    case 8: System.out.println("erreur de syntaxe"); break;
    default: System.out.println("action " + numAction + " non prevue");
  }
}
```

III - Programmation d'un automate fini déterministe

Résumé

Schéma d'interactions entre les classes pour les expressions (c.f. Moodle)



Dans le TP, faire :

- LexVelo par programmation directe
- AutoVelo, ActVelo par interpréteur de tables
- Interpréteur de tables fourni dans Automate

A - Programmation par automates finis

III - Programmation d'un automate fini déterministe

III.1 – Programmation directe

III.2 – Programmation par interpréteur de tables

III.3 – Traitement des erreurs dans l'analyse syntaxique

- **Erreurs au niveau de l'analyseur syntaxique**
  - **Erreur syntaxique = séquence non autorisée d'items lexicaux**
  - **Erreur sémantique = erreur au niveau des actions (par ex, valeur non autorisée pour un attribut lexical)**

- **Deux types de traitement** (pour erreurs syntaxiques et sémantiques)
  - **Erreurs fatales : passage dans l'état erreur avec arrêt définitif analyseur**
  - **Erreurs non fatales : passage dans l'état erreur, mais reprise de l'analyse "dès que possible"**
    - Concerne en général **données découpées en sous-parties** (par ex, suite d'expressions séparées par ';')
    - **Sous-partie erronée doit être ignorée** (par ex, l'expression erronée ne peut pas être affichée, ni participer au calcul d'un maximum)
    - **Au moins une transition permet de ressortir de l'état erreur sur un symbole de reprise** (par ex, sur PTVIRG car séparateur entre expressions)
    - Une **transition de reprise part de l'état erreur vers un état qui permet une reprise cohérente** de l'analyse (et doit prévoir les réinitialisations si nécessaire)

Exemple des expressions avec AutoExp, ActExp

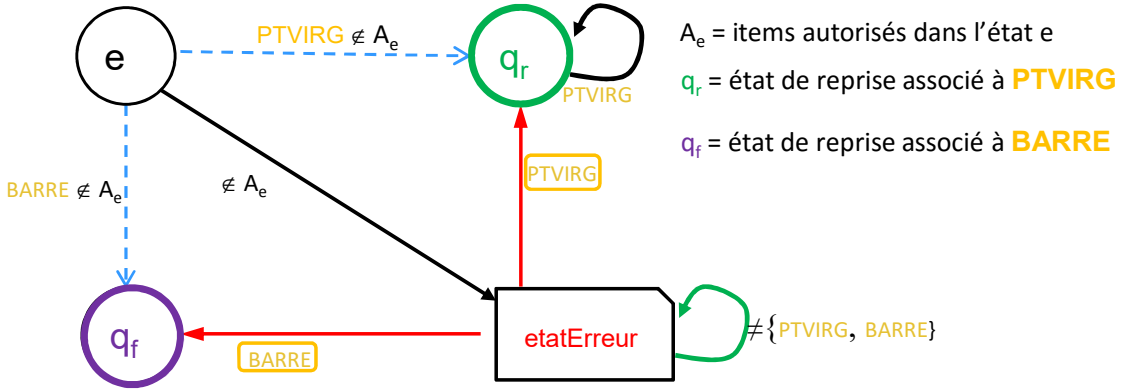
- **Jusqu'ici (et dans autoExpr.zip sous Moodle), toutes les erreurs étaient fatales**
- **Maintenant,** on souhaite distinguer erreurs fatales / non fatales
  - **Dans le cas d'erreurs syntaxiques : item lexical mal placé**
  - **Dans le cas d'erreurs sémantiques : détectées lors des actions**

III - Programmation d'un automate fini déterministe

III.3 – Traitement des erreurs dans l'analyse syntaxique

- **Dans le cas des erreurs syntaxiques**

- Erreurs non fatales = reprise après erreur
  - Symboles de reprise : par exemple **PTVIRG**, **BARRE**
  - **Détection erreur syntaxique : transition vers état erreur**
  - **Ajout** de transitions pour sortir de l'état erreur vers un état de **reprise, en fonction du symbole de reprise.**
  - **Mais si erreur syntaxique = symbole de reprise mal placé alors détection et reprise simultanée transition directe vers état de reprise (pas de passage par l'état erreur)**



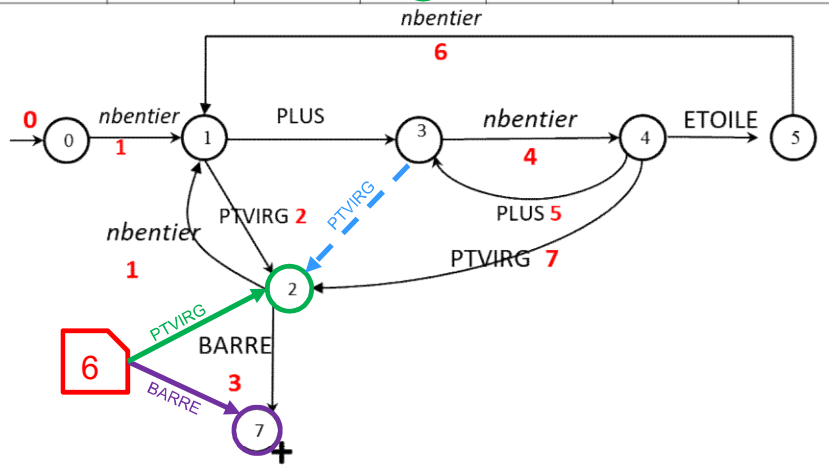
A_e = items autorisés dans l'état e

q_r = état de reprise associé à **PTVIRG**

q_f = état de reprise associé à **BARRE**

AutoExp.TRANSIT

Items / états	NBENTIER 0	PLUS 1	PTVIRG 2	ETOILE 3	BARRE 4	AUTRES 5
0	1	6	2	6	7	6
1	6	3	2	6	7	6
2	1	6	2	6	7	6
3	4	6	2	6	7	6
4	6	3	2	5	7	6
5	1	6	2	6	7	6
etatErreur=6	6	6	2	6	7	6



- **Dans le cas des erreurs sémantiques (actions)**
- 2 types :
  1. **Fatales :**
     Ex : saturation d'une structure de données ...
     - Arrêt de l'exécution → information pour interpréteur
  2. **Non fatales** :
     Ex : cours nul dans une cotation (non autorisé) ...
     - Passage à la fiche suivante dès que possible après la détection d'erreur, avec une **pseudo-transition** qui force l'état courant a l'état d'erreur.
     - La **sous-partie erronée doit être ignorée**
       Ex: l'exp erronée ne doit pas être affichée ni servir au calcul du max
- Utilisation d'une méthode ***erreur*** (ci-contre)
- Attention à la validité des modifications effectuées

  Uniquement quand sous-partie sans erreur!

- Dans Automate, on a donc en plus :

```
/** Types d'erreurs détectées */
protected static final int FATALE = 0;
protected static final int NONFATALE = 1;

/** Gestion des erreurs sémantiques
 * @param tErr type de l'erreur (FATALE ou NONFATALE)
 * @param messErr message associé à l'erreur
 */
protected final void erreur(int tErr, String messErr) {
  Lecture.attenteSurLecture(messErr);
  switch (tErr) {
    case FATALE: errFatale = true; break;
    case NONFATALE: etatCourant = etatErreur; break;
    default:
    Lecture.attenteSurLecture("paramètre incorrect");
  }
}
```