

22/05/07		Module : I810
		Compilation

# Compilation : Compte-rendu de TP

BERTHET Laurent		Compte Rendu de TP
CIMINERA Nicolas		
REY Charles		M1-TIC-ISC

15/05/07		Module : I810
 UNIVERSITE CHAMBERY ANNECY SAVOIE		Compilation

Table des Matières

Table des figures

## I. Cahier des charges

### 1. Enoncé de la tâche confiée

Pour ce TP il nous a été confié comme tâche de générer la table de reconnaissance syntaxique. Comme il a été vu en cours cette table permet de déterminer la validité d'une phrase à partir d'une grammaire fournie. Il existe plusieurs types de tables de reconnaissance syntaxique, mais afin de simplifier les choses nous devons réaliser un générateur de table *SLR* (ou *LRO*) avec une visualisation de la génération.

Afin de réaliser cette table nous avons choisi d'utiliser le langage Java, ce langage se prêtant bien à la manipulation de fichiers XML (utilisés dans la plupart des communications entre les différentes parties du TP) et à la manipulation des listes utilisées dans le cadre des tables.

### 2. Solution théorique adoptée

Afin de pouvoir générer cette table, nous avons suivi l'approche théorique vue en cours. C'est-à-dire que les différentes tables nécessaires nous étant fournies (tables des symboles, liste des productions de la grammaire, tables des séparateurs et table des fermetures), nous générons les 4 règles permettant d'obtenir les actions à partir de ces tables ainsi que les suivants de chaque symbole du langage. Ensuite nous construisons une table *SLR* vide contenant uniquement les séparateurs et les symboles en abscisse et le nombre d'état de la table des fermetures en ordonnée.

Une fois cette table générée nous commençons par appliquer les règles décaler et successeur à la table des fermetures afin d'obtenir les actions décaler et successeur nous arrivons ainsi à la table des transitions puis nous appliquons les règles restantes c'est à dire décaler et accepter et nous obtenons ainsi la table de reconnaissance complète. En ce qui concerne l'affichage nous avons naturellement décidé d'afficher la table *LRO* mais également un log dans lequel apparaîtront les résultats de l'application des différentes règles. L'utilisateur déclenchera l'application de chaque groupe de règle (décaler/successeur et réduire/accepter) via des boutons de l'interface afin de rendre la démarche adoptée plus visible à celui-ci.

## II. Analyse Organique

### 1. Ordinogramme de la répartition des tâches

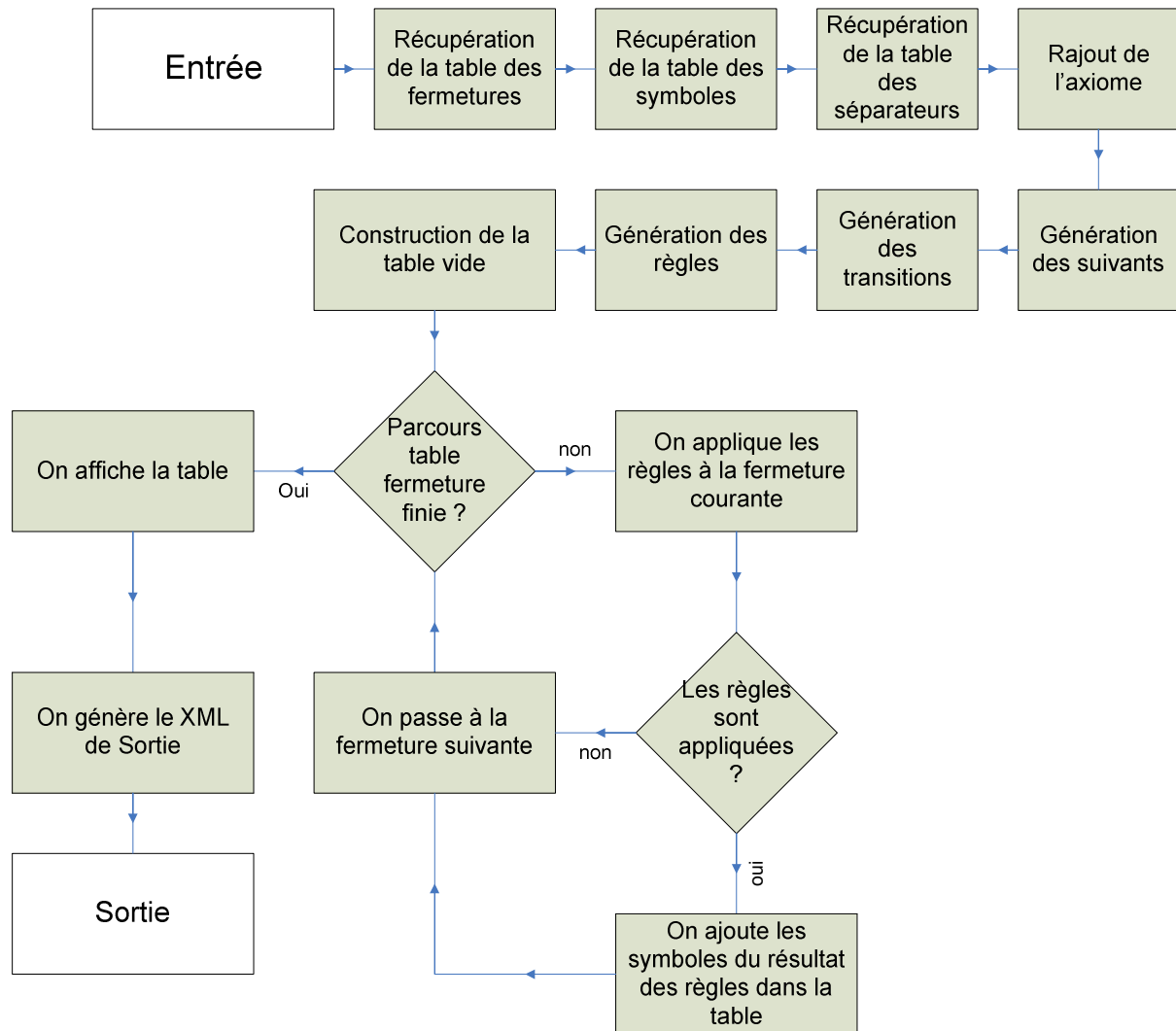


Figure 01 : « Ordinogramme de la réalisation »

Cet ordinogramme donne un point de vue global du projet. Les différentes tâches que l'on peut voir sont détaillées ci-dessous.

## 2. Constructeur de Table

Le constructeur de table est la partie qui va s'occuper de la génération de la table LR0. Elle va donc être constituée des différents éléments fournis en entrée via le XML (voir [partie XML](#)) et via le groupe précédent (voir rapport table des fermetures).

### a. Rajout de l'axiome

Dans cette étape la méthode concernée est `rajoutAxiome` de la classe `ConstructeurTable`.

Une fois les productions de la grammaire récupérées on doit rajouter un axiome afin de rendre l'analyse future compatible avec un symbole de délimitation d'entrée.

Dans ce but la première chose à faire est de récupérer le premier symbole non terminal de la partie gauche de la première production et de rajouter une production au début de la liste des productions constitué du symbole précédemment cité concaténé au caractère `.

Ensuite on doit rajouter dans la table des séparateurs le symbole terminal \$ permettant de délimiter les chaînes d'entrée.

Enfin, par défaut, il faut mettre comme unique suivant (voir ci-dessous) du symbole non terminal rajouté le symbole terminal \$.

### b. Génération des suivants

Pour cette étape les méthodes concernées sont `initSuivants()`, `reducteurs()` et `suivantsFrom()` de la classe `ConstructeurTable`.

Les données nécessaires à l'élaboration de la table LR0 étant récupérées, nous pouvons commencer les étapes préliminaires au remplissage de la table. La première d'entre elles concerne la génération des suivants.

Cette étape a pour but de construire les suivants en fonction de la grammaire fournie. Les suivants d'un symbole non terminal du langage sont les symboles terminaux du langage se situant à droite de celui-ci dans les développements des productions. De plus il faut prendre en compte les réductions c'est-à-dire que lorsque certains symboles non terminaux du langage se trouvent dans la partie droite d'une production, on dit qu'il se réduit au symbole de la partie gauche.

Exemple : Ici on peut dire que « T se réduit en E ».

$E \rightarrow T$

Dans le cas où un symbole se réduit en un autre symbole il faut alors ajouter tous les suivants du symbole qui réduit à la liste des suivants du symbole qui est réduit.

Nous avons choisi de représenter les suivants comme un dictionnaire avec en clé le symbole non terminal du langage et en valeur associée la liste des suivants de ce symbole sous forme de string (cf. attribut « suivants » de la classe constructeurTable).

Ensuite afin de déterminer les suivants de chaque symbole on procède en deux temps : tout d'abord on cherche les suivants simples de chaque symbole (sans ceux de leur éventuel réducteur), par la suite on cherchera les réducteurs des symboles et enfin on ajoutera les symboles des réducteurs au symboles concernés.

On commence donc par parcourir la liste des symboles non terminaux (méthode `initSuivants()`) et pour chacun de ces symboles on va chercher dans la liste des productions afin de trouver les symboles terminaux correspondants à la définition. On ajoute alors l'ensemble des suivants trouvés dans le dictionnaire suivants à la clé du symbole terminal que l'on vient d'avoir.

Ensuite avec la méthode `reducteurs()` on trouve les réducteurs des symboles. On parcourt la liste des productions dans l'ordre et pour chaque production qui correspond à une forme de réduction on stocke dans une liste le réducteurs et le réduit ce qui nous donne une liste contenant le réducteur et le réduit dans l'ordre d'apparition des symboles non terminaux.

Enfin on va pouvoir ajouter les suivants de chaque réducteur à la liste des suivants du symbole qu'il réduit dans le dictionnaire, en parcourant toujours dans l'ordre la liste des réducteurs, ce qui nous permet d'obtenir pour chaque symbole terminal du langage l'ensemble complet de ses suivants.

### c. Génération des transitions

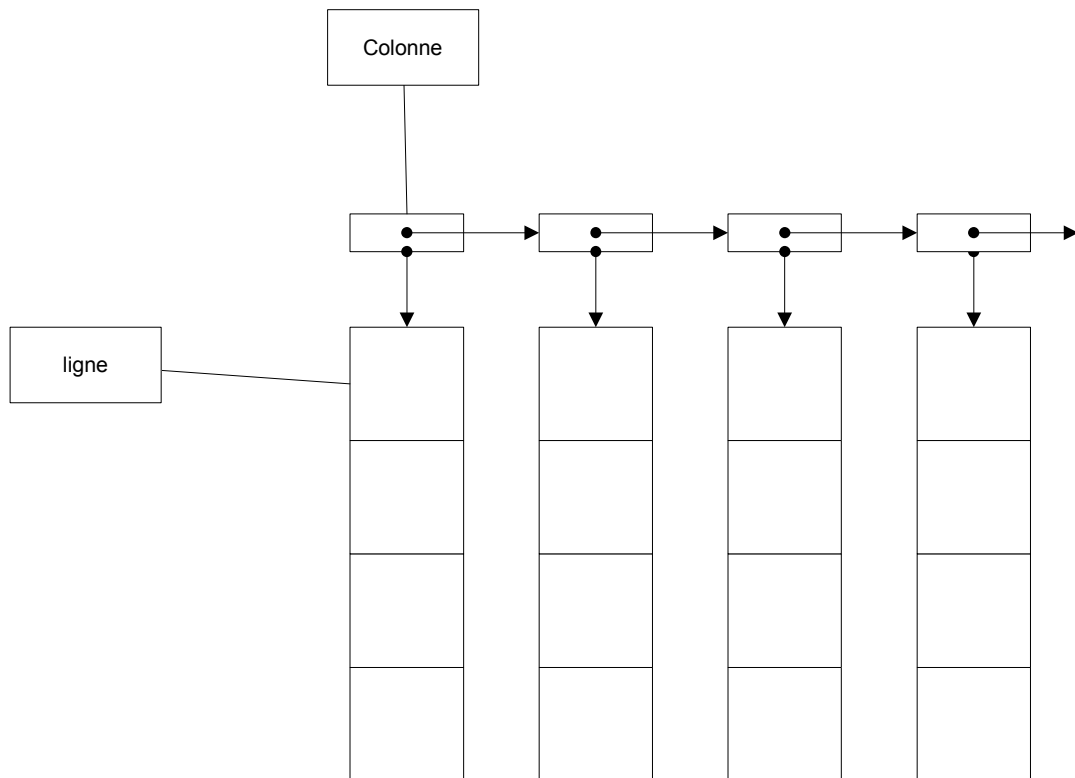
Les transitions nous sont fournies par le groupe générant la table des fermetures mais afin de pouvoir être exploitées par les règles qui en ont besoin nous devons leur faire subir une transformation en effet les règles ont besoin de savoir dans quel indice de colonne de la table des fermetures se trouve chaque transition.

Afin de simplifier cette tâche nous générons un dictionnaire avec comme clé les différentes transitions et comme valeurs l'indice de la colonne de la table des fermetures dans laquelle elle se trouve. Pour effectuer cette transformation nous nous servons de la méthode `initTransitions()` de la classe constructeurTable.

Cette méthode va parcourir chaque colonne de la table des fermetures et pour chaque transition de cette colonne créer une entrée avec la transition comme clé et l'indice de la colonne comme valeur

#### d. Construction de la table vide

Nous avons choisi de représenter la table vide par une liste de liste de String. Le format de sortie étant du XML le type string est le type idéal. Cette liste est représentée par l'attribut tableLR0 de la classe ConstructeurTable. Voici un schéma de l'architecture de la table.



Afin d'initialiser cette table nous utilisons la méthode initLR0. Cette méthode va tout d'abord créer une liste de liste de string dont la taille de la première liste est celle du nombre de séparateurs ajouté au nombre de symboles non terminaux du langage plus un afin de laisser la place pour les intitulés. Ensuite pour chaque liste de la première liste nous créons

une liste dont la taille est celle du nombre d'état plus un. La première liste de la liste étant les intitulés des états nous pouvons la remplir à partir de la table des fermetures en laissant une première case vide afin de laisser la place pour les intitulés des colonnes. Enfin pour chaque autre liste nous rentrons dans la première case le nom du séparateur ou du symbole non terminal.

#### e. Application des règles et remplissage de la table

Une fois la table initialisée nous pouvons la remplir. Cette tâche s'effectue grâce aux méthodes `remplirLR0WithRule()` de la classe `ConstructeurTable`. Cette méthode prend en paramètre la liste des règles à appliquer afin de ne pouvoir appliquer que les règles choisies.

Le fonctionnement en est simple, mais il faut au préalable avoir initialisé les règles (cf. [partie règles](#)). On parcourt alors les fermetures contenues dans la table des fermetures. Pour chaque fermeture on applique chaque règle de la liste passée en paramètre. L'application d'une règle nous renvoie liste de `Case` (cf. [partie règles](#)) contenant chacune un booléen indiquant si la règle peut être appliquée, les coordonnées de la case de la table de LR0 concernée, et enfin le symbole à mettre dans la case.

On parcourt alors la liste et pour chaque case retournée on vérifie si la règle a été acceptée et si oui, on peut mettre le symbole dans les coordonnées retournées. On passe ensuite à la fermeture suivante et ainsi de suite.

### 3. Règles

#### a. Principe

La Table LR0 est construite grâce à l'application de quatre règles sur les fermetures contenues dans la table des fermetures. Nous avons donc créé une classe pour chaque règle soit quatre classes.

Pour faciliter la gestion des règles, nous avons créé une interface `IRegle` qui contient les méthodes qui seront communes à toutes les règles.

Il y a trois méthodes communes :

■ **`ArrayList<Case> appliquerRegle(ArrayList<XMLElement> fer, int col)`**

Cette méthode prend en paramètre, une fermeture qui va être analysée et le numéro de la colonne d'où elle provient. Cette méthode retourne un `ArrayList` de `Case` (voir explication plus bas). Nous

expliquerons plus en détail le fonctionnement de cette méthode un peu plus bas.

### ■ **ArrayList<String> parseItem(ArrayList<XMLElement> xml)**

Cette méthode permet de mettre sous une forme adéquate une fermeture. Pour plus de simplicité, nous travaillons avec des fermetures sous la forme **[E, ->, E, ,, +, T]**. La méthode `parseItem` renvoie ce genre de liste.

### ■ **int getType()**

Cette méthode permet de connaître le type de la règle, c'est-à-dire si cette règle est une règle accepter, décaler, réduire ou successeur.

Toutes les règles ont de nombreux attributs identiques :

### ■ **ArrayList<String> patron**

Cet attribut est le modèle de la règle. On l'instancie lors de la construction de la règle.

### ■ **ArrayList<String> item**

Item correspond à la fermeture à laquelle on va essayer d'appliquer la règle. Cet attribut est la version transformée de la fermeture, on l'obtient en appliquant la méthode ***parseItem***.

### ■ **HashMap<ArrayList<XMLElement>, String> transitions**

C'est un dictionnaire de toutes les transitions provenant de la table des fermetures. C'est un dictionnaire qui a comme clé une transition et comme valeur l'index de la colonne de cette table.

Cet attribut est récupéré lors de la création de la règle.

### ■ **HashMap<String, ArrayList<String>> suivants**

C'est un dictionnaire contenant la liste des tous les suivants de tous les symboles. La clé de ce dictionnaire est le symbole. Et donc la valeur est la liste de ses suivants.

Cet attribut est récupéré lors de la création de la règle.

### ■ **ArrayList<String> listeSeparateurs**

Contient la liste des tous les séparateurs du langage. Cet attribut est récupéré lors de la création de la règle.

### ■ **ArrayList<String> listeSymboleNonT**

Contient la liste des symboles non terminaux du langage.  
Cet attribut est récupéré lors de la création de la règle.

Voici un exemple de création de règle avec la règle 4 (successeur) :

```
public Regle4(HashMap<ArrayList<XMLElement>, String> t,
HashMap<String, ArrayList<String>> s, ArrayList<String> INonT,
ArrayList<String> ls) {
    patron = new ArrayList<String>();
    patron.add("A");
    patron.add("->");
    patron.add("alpha");
    patron.add(".");
    patron.add("B");
    patron.add("beta");

    this.listeSeparateurs = ls;
    this.listeSymboleNonT = INonT;
    this.transitions = t;
    this.suivants = s;
}
}
```

Dans la première partie du constructeur, on initialise le patron qui va nous permettre de comparer avec une fermeture.

La seconde partie permet de récupérer les dictionnaires.

La construction des autres règles se fait de la même façon à part l'initialisation du patron qui diffère pour chaque règle.

## b. Fonctionnement de l'application des règles

Afin de clarifier la manière dont nous avons procédé afin de pouvoir appliquer les règles voici les explications de la méthode **appliquerRegle** :

Cette méthode retourne une liste de Case. Une case contient plusieurs informations :

### ■ **Boolean applique**

Ce booléen permet de savoir si on peut appliquer la règle avec cette fermeture. Si c'est le cas, les autres attributs auront des valeurs significatives, sinon on ne se servira pas des autres attributs.

### ■ String symbole

Cet attribut contient la chaîne correspondante à l'application d'une règle, c'est-à-dire, r2, d6, acc ou 1suivant le type de règle appliquée.

### ■ int ligne

C'est le numéro de la ligne dans laquelle on doit mettre le symbole.

### ■ int colonne

C'est le numéro de la colonne où l'on doit insérer le symbole

La méthode ***appliquerRegle*** retourne une liste de **Case** contenant une seule **Case**. Lorsqu'on applique une fermeture avec la regle3 qui la règle concernant les décalages, cette méthode peut retourner plusieurs **Case**.

Avant de commencer à analyser la fermeture, nous la transformons grâce à la méthode *parseItem*. Une fois cette transformation effectuée nous pouvons commencer à comparer la fermeture (item) avec notre patron. Si on s'aperçoit qu'on peut appliquer la règle à la fermeture, on crée une case avec les bons paramètres. Dans le cas contraire, on crée une case dans laquelle on met *false* dans le booléen pour indiquer que la fermeture ne convient pas.

La méthode *appliquerRegle* est différente pour toutes les règles. Celle de la règle 1 et de la règle 4 sont pratiquement identique car les deux règles fonctionnent de la même façon, les patrons sont les mêmes. La seule différence est que dans le patron ( $A \rightarrow \alpha \bullet a \beta$ ), le « a » ne correspond pas à la même chose, dans la règle décaler, « a » doit être un symbole terminal de la grammaire alors que dans la règle successeur, « a » doit être un symbole non terminal de la grammaire. Pendant la transformation de la fermeture en item, il a fallu aussi penser que le  $\alpha$  peut être vide donc il a fallu ajouter un élément vide dans item quand  $\alpha$  est vide pour que l'on puisse comparer au patron.

La méthode de la règle acceptation a été la plus simple à réaliser car il n'y a pas beaucoup de test à effectuer. Il suffit de regarder si l'item correspond à l'axiome rajouté. Pour ce faire il faut regarder si le symbole qui est à gauche de la flèche est le même que celui qui est à droite.

Pour la règle réduction, il a fallu faire attention à ne pas inclure la règle successeur car les patrons ont presque la même forme ( $A \rightarrow a \bullet$ ) et ( $A' \rightarrow A \bullet$ ). Dans cette règle, on parcourt la liste des suivants A pour retourner toutes les cases applicables à la fermeture.

## 4. Récupération et Génération du XML

### a. Récupération des données

Pour notre application, nous avons besoin de la liste des symboles, de la liste des séparateurs, de la table des fermetures et des productions. Nous utilisons deux méthodes différentes pour récupérer ces ressources.

☞ Pour les productions, la liste des séparateurs et la liste des symboles, nous recevons un fichier XML dont voici la DTD :

```
<!DOCTYPE grammaire [
  <!ELEMENT REGLE (ELEMENT|OPTION)*>
  <!ELEMENT liste_mots_clefs (mot_clef)*>
  <!ELEMENT ELEMENT (#PCDATA)>
  <!ELEMENT grammaire
liste_mots_clefs|liste_symboles|liste_separateurs|productions)*>
  <!ELEMENT liste_separateurs (separateur)*>
  <!ELEMENT productions (REGLE)*>
  <!ELEMENT mot_clef (#PCDATA)>
  <!ELEMENT symbole (#PCDATA)>
  <!ELEMENT liste_symboles (symbole)*>
  <!ELEMENT separateur (#PCDATA)>
  <!ELEMENT OPTION (ELEMENT)*>
1>
```

### b. Données intermédiaires

Notre application engendre différents résultats, sous 3 formes différentes :

#### ● Graphique

Nous affichons la table LR0 sous forme de tableau :

	id	+	*	(	)	\$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

#### ● Textuelle

Nous affichons également les pas de calculs :

```

debut a Wed May 23 17:23:43 CEST 2007
exempleprof.xml chargé
liste des symboles
E, T, F,
liste separateurs
id, +, *, (, ),
productions
E->E+T
E->T
  
```

#### ■ Fichier Texte

Nous assemblons tous les messages que l'application lance au cours de son exécution, ces messages contiennent tous les pas de calculs effectués pour construire la table des fermetures ainsi que ceux de la table LR0. Le fichier qui en résulte se trouve dans le même répertoire que le programme il se nomme « resultat.txt ».

#### c. Génération du XML

Une fois que la table LR0 est générée, elle est exportée en XML ce document XML respecte la DTD suivante :

```

<!DOCTYPE LR0 [
  <!ELEMENT LR0 (symbol)*>
  <!ELEMENT etat (#PCDATA)>
  <!ATTLIST etat
    nom_etat CDATA #REQUIRED>
  <!ELEMENT symbol (etat)*>
  <!ATTLIST symbol
    nom_symbol CDATA
    #REQUIRED
    type CDATA #REQUIRED>
  ]
  
```

Voici un exemple de fichier XML généré par notre application correspondant à la table de LR0 ci-dessus :

```

<?xml version="1.0" encoding="UTF-8"?>
<LR0>
  <symbol nom_symbol="id" type="separateur">
    <etat nom_etat="1">d5</etat>
    <etat nom_etat="2" />
    <etat nom_etat="3" />
    <etat nom_etat="4" />
    <etat nom_etat="5">d5</etat>
    <etat nom_etat="6" />
  </symbol>
  
```



```

<etat nom_etat="7">d5</etat>
<etat nom_etat="8">d5</etat>
<etat nom_etat="9" />
<etat nom_etat="10" />
<etat nom_etat="11" />
<etat nom_etat="12" />
</symbol>
<symbol nom_symbol="+" type="separateur">
  <etat nom_etat="1" />
  <etat nom_etat="2">d6</etat>
  <etat nom_etat="3">r2</etat>
  <etat nom_etat="4">r4</etat>
  <etat nom_etat="5" />
  <etat nom_etat="6">r6</etat>
  <etat nom_etat="7" />
  <etat nom_etat="8" />
  <etat nom_etat="9">d6</etat>
  <etat nom_etat="10">r1</etat>
  <etat nom_etat="11">r3</etat>
  <etat nom_etat="12">r5</etat>
</symbol>
<symbol nom_symbol="*" type="separateur">
  <etat nom_etat="1" />
  <etat nom_etat="2" />
  <etat nom_etat="3">d7</etat>
  <etat nom_etat="4">r4</etat>
  <etat nom_etat="5" />
  <etat nom_etat="6">r6</etat>
  <etat nom_etat="7" />
  <etat nom_etat="8" />
  <etat nom_etat="9" />
  <etat nom_etat="10">d7</etat>
  <etat nom_etat="11">r3</etat>
  <etat nom_etat="12">r5</etat>
</symbol>
<symbol nom_symbol="(" type="separateur">
  <etat nom_etat="1">d4</etat>
  <etat nom_etat="2" />
  <etat nom_etat="3" />
  <etat nom_etat="4" />
  <etat nom_etat="5">d4</etat>
  <etat nom_etat="6" />
  <etat nom_etat="7">d4</etat>
  <etat nom_etat="8">d4</etat>
  <etat nom_etat="9" />
  <etat nom_etat="10" />
  <etat nom_etat="11" />

```



```

    <etat nom_etat="12" />
</symbol>
<symbol nom_symbol=")" type="separateur">
    <etat nom_etat="1" />
    <etat nom_etat="2" />
    <etat nom_etat="3">r2</etat>
    <etat nom_etat="4">r4</etat>
    <etat nom_etat="5" />
    <etat nom_etat="6">r6</etat>
    <etat nom_etat="7" />
    <etat nom_etat="8" />
    <etat nom_etat="9">d11</etat>
    <etat nom_etat="10">r1</etat>
    <etat nom_etat="11">r3</etat>
    <etat nom_etat="12">r5</etat>
</symbol>
<symbol nom_symbol="$" type="separateur">
    <etat nom_etat="1" />
    <etat nom_etat="2">acc</etat>
    <etat nom_etat="3">r2</etat>
    <etat nom_etat="4">r4</etat>
    <etat nom_etat="5" />
    <etat nom_etat="6">r6</etat>
    <etat nom_etat="7" />
    <etat nom_etat="8" />
    <etat nom_etat="9" />
    <etat nom_etat="10">r1</etat>
    <etat nom_etat="11">r3</etat>
    <etat nom_etat="12">r5</etat>
</symbol>
<symbol nom_symbol="E" type="variable">
    <etat nom_etat="1">1</etat>
    <etat nom_etat="2" />
    <etat nom_etat="3" />
    <etat nom_etat="4" />
    <etat nom_etat="5">8</etat>
    <etat nom_etat="6" />
    <etat nom_etat="7" />
    <etat nom_etat="8" />
    <etat nom_etat="9" />
    <etat nom_etat="10" />
    <etat nom_etat="11" />
    <etat nom_etat="12" />
</symbol>
<symbol nom_symbol="T" type="variable">
    <etat nom_etat="1">2</etat>
    <etat nom_etat="2" />

```



```
<etat nom_etat="3" />
<etat nom_etat="4" />
<etat nom_etat="5">2</etat>
<etat nom_etat="6" />
<etat nom_etat="7">9</etat>
<etat nom_etat="8" />
<etat nom_etat="9" />
<etat nom_etat="10" />
<etat nom_etat="11" />
<etat nom_etat="12" />
</symbol>
<symbol nom_symbol="F" type="variable">
  <etat nom_etat="1">3</etat>
  <etat nom_etat="2" />
  <etat nom_etat="3" />
  <etat nom_etat="4" />
  <etat nom_etat="5">3</etat>
  <etat nom_etat="6" />
  <etat nom_etat="7">3</etat>
  <etat nom_etat="8">10</etat>
  <etat nom_etat="9" />
  <etat nom_etat="10" />
  <etat nom_etat="11" />
  <etat nom_etat="12" />
</symbol>
</LR0>
```

### III. Analyse Fonctionnelle

La table LR0 sera utilisée au cours de la phase de reconnaissance syntaxique, afin de vérifier la validité syntaxique des phrases à analyser. Tous les traitements de notre programme sont effectués à partir de String ou de types composés à partir de ce type.

Nous affichons un message d'erreur lors du chargement pour indiquer que le fichier sélectionné n'est pas un fichier XML

## IV. Bilan

Pour réaliser la table LR0, nous avons dû dans une première partie comprendre le fonctionnement de remplissage de cette table c'est-à-dire comprendre les quatre règles pour arriver à les appliquer. Une fois que l'on a compris le mécanisme de remplissage, il a fallu choisir notre implémentation. Après avoir bien choisi notre modèle, nous nous sommes repartis les tâches pour que lors de la mise en commun il y ait le moins de retouches à faire du point de vue entête de fonction. La chose qui nous a posé quelques problèmes a été de se mettre d'accord avec tous les autres groupes des besoins de chacun.

Pour améliorer la construction de la table LR0, il faudrait améliorer le parcours de la table des fermetures c'est-à-dire qu'il faudrait réduire le nombre de passage dans la table pour voir si une règle est applicable à une fermeture.